Article

Optimization Techniques for a Distributed In-Memory Computing Platform by Leveraging SSD

June Choi ¹, Jaehyun Lee ¹, Jik-Soo Kim ² and Jaehwan Lee ^{1,*}

¹ School of Electronics and Information Engineering, Korea Aerospace University, Goyang-si 10540, Korea; wnsdl1210@gmail.com (J.C.); aprilmagnificat@gmail.com (J.L.)

² The Department of Computer Engineering, Myongji University, Yongin-si 03674, Korea; jiksoo@mju.ac.kr

* Correspondence: jlee@kau.ac.kr; Tel.: +82-2-300-0122

Abstract: In this paper, we present several optimization strategies that can improve the overall performance of the distributed in-memory computing system, “Apache Spark”. Despite its distributed memory management capability for iterative jobs and intermediate data, Spark has a significant performance degradation problem when the available amount of main memory (DRAM, typically used for data caching) is limited. To address this problem, we leverage an SSD (solid-state drive) to supplement the lack of main memory bandwidth. Specifically, we present an effective optimization methodology for Apache Spark by collectively investigating the effects of changing the capacity fraction ratios of the shuffle and storage spaces in the “Spark JVM Heap Configuration” and applying different “RDD Caching Policies” (e.g., SSD-backed memory caching). Our extensive experimental results show that by utilizing the proposed optimization techniques, we can improve the overall performance by up to 42%.

Keywords: Apache Spark; memory management; solid-state drive; in-memory processing framework; performance; PageRank; transitive closure; TeraSort; k-means clustering; Java Virtual Machine heap configuration; resilient distributed dataset

Citation: Choi, J.; Lee, J.; Kim, J.-S.; Lee, J. Optimization Techniques for a Distributed In-Memory Computing Platform by Leveraging SSD. *Appl. Sci.* **2021**, *11*, 8476. <https://doi.org/10.3390/app11188476>

Academic Editors: Manuel Armada and Arcangelo Castiglione

Received: 21 June 2021

Accepted: 9 September 2021

Published: 13 September 2021

Publisher’s Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2021 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

As the big data industry develops rapidly, there have been a number of research efforts to build distributed processing frameworks, such as MapReduce [1] of Hadoop [2], that can effectively store and process “Big Data”. However, the performance of Hadoop based on normal spindle disks (HDD) can be degraded due to the Hadoop distributed file system’s (HDFS) [3] read/write operations, especially for machine learning workloads where there can be a lot of iterative jobs and intermediate data. To address this problem, the Spark [4] framework was introduced, which can effectively cache the intermediate data to memory so that the cluster computing platform can dramatically improve the overall performance.

However, according to a comprehensive study analyzing Spark’s performance behaviors [5], Spark still can have performance degradation problems due to some straggler tasks. As a result of analyzing the tasks that affect the whole job completion time of Spark, *garbage collection*, *shuffle write*, and *shuffle read* are identified as main factors that can negatively affect the performance of the Spark system. Unfortunately, a detailed analysis of the main reasons why these aforementioned tasks affect the task processing in Spark and potential solutions have not yet been thoroughly investigated.

In this paper, we first analyze the specific factors that can explain why garbage collection, shuffle write, and shuffle read affect the task processing in Spark and present associated situations and possible solutions through extensive experiments on a Spark cluster. Specifically, to analyze the factors causing the degradation of “whole job completion time” of the Spark, we performed various experiments with PageRank [6], transitive closure [7], TeraSort [8], and k-means clustering [9] workloads. Based on our extensive

experimental results, we found potential performance degradation factors in the Spark system that can be summarized as follows:

1. **The performance degradation on Java garbage collection:** Since Spark is running on Java Virtual Machines (JVM), Java garbage collections can especially occur when there is a lack of Spark executor memory, i.e., JVM heap size.
2. **The performance degradation on shuffle spill:** When the shuffle write is being processed, if the shuffle memory of the Spark executor memory (JVM heap) is insufficient, Spark will spill the shuffle data out to the disk (HDD). In this case, Spark has to serialize the data to write and de-serialize to read the data from the disk. Since a CPU resource is required for the serialization and de-serialization processes, this can slow down the overall processing of tasks.
3. **The performance degradation on shuffle read blocked time:** As we will see from the experimental results, if the number of tasks keeps increasing in iterative stages as in the transitive closure workload, there can be scheduling overhead and Java garbage collections due to lack of Spark executor memory. This can cause shuffle read tasks to be blocked which can result in poor performance.

The main contribution of this paper is that we propose effective Spark cluster configuration strategies that can actually improve the overall system performance by utilizing SSDs in order to overcome the physical memory limits of the cluster. In a typical cluster computing environment consisting of commodity servers, it would be difficult to set up a large amount of main memory. Therefore, we address the performance degradation problems in a distributed in-memory computing system that can occur due to insufficient main memory amounts by effectively leveraging SSDs. Our optimization strategy is two-fold as follows.

First, we change the capacity fraction ratios of the shuffle and storage spaces in the “Spark JVM Heap Configuration”. According to the experimental results of different workloads, we observe the performance differences depending on the workload’s memory usage patterns.

Second, we apply different “RDD Caching Policies” such as no cache, memory-only cache, disk-only cache, and SSD-backed memory cache. In most cases, the SSD-backed memory caching policy shows the best performance unless all of the RDDs can completely fit in the actual main memory.

We conducted an empirical performance evaluation under various configurations and different workloads. Our experimental results show that by carefully allocating the amounts of storage and shuffle areas in the Spark’s JVM heap based on the memory usages of target workloads and by applying an optimal RDD caching policy, we can substantially reduce the total execution time by up to 42%.

The rest of this paper is structured as follows. In Section 2, we briefly describe the background for the Spark system and present related work, and Section 3 presents the use of Spark and the configuration of the Spark cluster and details our optimization methodology to improve the overall performance. In Section 4, we present our experimental results and analysis on the factors of the performance degradation and associated solutions for them. Section 5 discusses evaluation results and summarizes our findings, and we conclude and discuss future work in Section 6.

2. Background and Related Research Work

2.1. Background

Apache Hadoop has been the de facto standard “big data” storage and processing platform by effectively distributing and managing the data and computations across many nodes. However, Hadoop can not achieve competitive performance for some applications such as machine learning, especially those consisting of several iterative stages and a relatively large amount of intermediate data. This is because in every iterative stage, Hadoop has to read and write data from/to an HDFS generated by MapReduce.

Apache Spark exploits *resilient distributed datasets* (RDD) [10] that can effectively manage any intermediate/final data in the main memory such as caching that can be efficiently utilized in each stage of iterative applications. Since the RDD is immutable, Spark introduces a concept of *lineage* that can keep track of the history of RDD creations, which can be used for failure recovery. Through this concept, Spark can reduce the number of I/O operations on the disk compared to the Hadoop. Because of this distributed in-memory computing capability, Spark typically shows better performance than Hadoop for a wide range of data analytics applications.

However, the RAM used for the main memory to store Spark's data is relatively expensive in terms of unit price per byte, so that it would be very difficult to construct a large enough amount of RAM in the Spark cluster to support various workloads. Therefore, the limited capacity of RAM can restrict the overall speed of Spark processing. If Spark cannot cache RDD to RAM due to limited space during the application processing, Spark has to re-generate the missing RDDs which could not fit into the RAM in every stage, becoming similar to Hadoop's approach. In addition, because the Spark job is a Java process running on the JVM, GC (garbage collection) occurs whenever the available amount of memory is limited. Since RDD is typically cached on the old space of JVM, when a major GC occurs, it can substantially affect the whole job processing performance. Moreover, the lack of memory can cause "Shuffle Spill", which is the process of spilling the intermediate data generated during shuffle from memory to disk. shuffle spill involves many disk I/O operations and CPU overheads. Consequently, a new solution must be considered in order to cache all of the RDDs and to secure memory for shuffle.

2.2. Related Work

There have been many related studies in the literature regarding performance improvements of the Spark platform as follows. Table 1 summarize related work according to the subjects.

Table 1. Summary of related work.

Categories	Improvement Methods
Spark Shuffle Improvement	Network and Block Optimization [11–13] Cost-Effectiveness [14]
Performance Analysis, Modeling	I/O-aware Analytical Model [15] Empirical Performance Model [16]
Parameter Tuning	Empirical Tuning [17,18] Auto-Tuning [19]
Memory Optimization	Memory Optimization [20]
JVM and GC Overhead	JVM Overhead [21] GC Overhead [22]
Cache Management Policy	RDD Policy [23]

- Improving performance of Spark shuffle:** The optimization of shuffle performance in Spark [11] analyzes the bottleneck on running a Spark job and presents two alternatives, columnar compression and shuffle file consolidation. Because spilling all data of the in-memory buffer is a burden for the OS, the solution is to write fewer, larger files in the first place. Nicolae et al. presented a new adaptive I/O method for collective data shuffling [12]. They adapt the accumulation of shuffle blocks to the individual rate of processing for each reducer task, while coordinating the reducers to collaborate in the optimal selection of the sources (i.e., where to fetch shuffle blocks from). In this way, they balance loads well and avoid stragglers with reducing the memory usage for buffer.

Riffle [13] is one of the most efficient shuffle services for large-scale data analytics. Riffle merges fragmented intermediate shuffle files into larger block files and thus converts small, random disk I/O requests into large, sequential ones. Riffle also mixes both merged and unmerged block files to minimize merge operation overhead.

Pu et al. suggest a cost-effective shuffle service by combining cheap but slow storage with fast but expensive storage to achieve good performance [14]. They run TPC-DS, CloudSort, and Big Data Benchmark on their system and show a reduction of resource usage by up to 59%.

They all emphasize the improvement of shuffle performance and cost-effectiveness by coordinating network transfer or reducing disk I/O. However, in our study, we configure the JVM heap to reduce shuffle spill which is a bottleneck in the shuffle phase and job completion time.

- Performance analysis, modeling, and optimization for Spark:** The authors of [15] show that storage I/O plays a heavy role in the in-memory cluster computing frameworks and propose an I/O-aware analytical model to reason through the performance of Spark programs. The proposed model can analytically explain and predict the run-time behavior of iterative algorithms that are computation/shuffle-heavy algorithms. They also apply the proposed model on cost optimization in Google Cloud. Marcu et al. show the performance analysis of Spark and Flink based on their experimental results using representative workloads in a comparative manner [16]. They identify a set of the four most important parameters having a major influence on performance. The task parallelism, the network behaviour during the shuffle phase, the memory, and the data serialization are the most important parameters they pick. On the other hand, in our study, we focus on the performance improvement methodology by appropriately choosing the type of *storage* and allocating the amount of storage and shuffle areas in the *Spark's JVM heap* according to the memory usage patterns of target workloads.
- Parameter Tuning for Spark:** There have been some research trials to improve the performance of Spark by tuning all configuration parameters. Petridis et al. show their experience of parameter tuning of Spark by trial and error [17]. They pick 12 key application instance-specific parameters and assess their impact using real executions on a petaflop supercomputer. Similarly, Gounaris and Torres [18] analyze the impact of the most important tunable Spark parameters concerning shuffling, compression, and serialization on the application performance in an empirical manner. They provide extensive experimental results on the Spark-enabled Marenstrum III (MN3) computing infrastructure of the Barcelona Supercomputing Center. In contrast to the empirical tuning method, Yu et al. suggest an auto-tuning scheme for an in-memory computing platform [19]. They take the size of the input dataset and 41 configuration metrics as parameters of the performance model. They use hierarchical modeling (HM) to combine a number of individual sub-models in a hierarchical manner and use the genetic algorithm (GA) to search for the optimal configuration. Although the above research works try to reach the optimal performance by tuning parameters of Spark, which is similar to our work, our paper is different from these because we use an SSD-backed memory cache policy to effectively extend the physical memory limitation of a Spark cluster.
- Memory optimization for MapReduce-based data processing:** The authors of [20] analyze in depth the impact of memory efficiency on the performance of the Hadoop-compatible Flame-MR framework. They present several memory optimization techniques to reduce the amount of object allocations and deallocations, decreasing GC overheads and the overall execution time. In our study, we leverage SSDs for improving the performance on the in-memory system.
- Reducing JVM and garbage collection overhead in Spark:** JVM and GC comprise one of the major overheads in the Spark platform, especially when the workload

suffers from memory limitations. Lion et al. point out that JVM warm-up overhead is one of the major bottlenecks in HDFS, Hive, and Spark platforms [21]. They propose a new JVM that amortizes the warm-up overhead by reusing a pool of already warm JVMs.

Maas et al. find that GC-induced pauses can have a significant impact on Spark [22]. Thus, they propose a holistic runtime system, a distributed language runtime that collectively manages runtime services to coordinate GC-induced pauses across multiple nodes.

Even though both papers deal with JVM- and GC-related issues in Spark for general cases, our paper assumes that workloads suffer from memory limitations.

- **Cache management policy optimization for Spark:** The authors of [23] propose least composition reference count (LCRC), a dependency-aware cache management policy that considers both intra-stage and inter-stage dependency. LCRC can rewrite these inter-stage accessed blocks into memory before its next use. In our study, we leverage SSDs rather than enhancing the cache policy for improving the performance of the in-memory system.

3. Optimization Techniques for the Spark Platform

In this section, we present our cluster environment and associated optimization techniques that can improve the overall performance of the Spark platform.

3.1. Cluster Environment

Figure 1 shows our testbed cluster consisting of one namenode (master) and four datanodes (slaves). In the namenode (master), we configured the *NameNode* and *Secondary NameNode* of Hadoop (HDFS) and the *Driver Node* (master node) of Spark. In each datanode, we run the *DataNode* of Hadoop (HDFS) and *Worker Node* of Spark. The namenode and datanode machines have the same H/W environments (3.4 GHz Xeon E3-1240V3 Quad-Core Processor with hyper-threading), except for the amount of main memory (8 GB for the namenode and 4 GB for each datanode). We used two SSDs as storage spaces where a 120 GB SATA3 SSD is used for the operating system, and a 512 GB SATA3 SSD is equipped for the HDFS, respectively. In addition, the 512 GB SATA3 SSD can be effectively leveraged for expanding the bandwidth of insufficient main memory to cache the RDDs of Spark. All nodes including namenode and datanode are connected with a 1 Gb Ethernet switch, as seen from Figure 1. Table 2 shows the summary of hardware and software configurations in each datanode of our testbed cluster.

Figure 1. The testbed cluster for experiments.

Table 2. H/W and S/W configurations of a datanode.

Parts	Specifications
CPU	Intel Xeon E3-1240V3
Memory	4 GB DRAM(DDR3-1600 Mhz ECC)
Storage	120 GB SSD SATA 512 GB SSD SATA
Software	CentOS 6.6 Java Virtual Machine: OpenJDK 1.7.0 Hadoop: Apache Hadoop 2.6.2 Spark: Apache Spark 1.5.2

3.2. Spark JVM Heap

A Spark job runs as a Java process on the Java Virtual Machine (JVM), and Spark exploits the Scala, a functional language extended from Java. The worker process of Spark also runs on the JVM of each datanode, so that on each datanode, the worker process has the JVM heap in the main memory as depicted in Figure 2. When Spark submits a job, the worker process that has the JVM heap executes the job as distributed tasks.

Figure 2. Spark JVM heap.

We can customize the ratio of the JVM heap size of a Spark worker through the configuration file `spark-defaults.conf` in the `spark/conf/` directory. In the `spark-defaults.conf` file, the value of `spark.executor.memory` is the JVM heap size where the default is 512 MB that each worker node can utilize in the datanode. In addition, the value of `spark.storage.safetyFraction` is fixed as 0.9, which means that Spark can actually use up to 90% of JVM heap size (also known as safety area). This is for preventing the JVM from generating OOM (out of memory) errors due to the lack of available main memory during the task processing.

In this safety area, the overall JVM heap space is divided into three sub-regions: *unroll*, *storage*, and *shuffle* spaces, as we can see from Figure 2. The unroll space is used for unrolling data blocks in memory. When an RDD is cached on other storage media such as an SSD or HDD not on the main memory, the RDD should be *serialized*. Then, when Spark reads this RDD back to the memory, the RDD has to be *unrolled*. The storage space is used for *caching* an RDD. If the storage space is not sufficient for caching the RDD, some RDDs can be evicted from this space based on the LRU (least recently used) policy, or they can be cached on other storage media, such as an SSD. The shuffle space is used for *shuffling* the intermediate data. For iterative applications such as machine learning, this shuffle space can play an important role since it can substantially affect the overall job completion time.

In the default Spark configuration, the storage and shuffle spaces of the JVM heap have capacity fraction ratios of 0.6 and 0.2, respectively (i.e., 60% of safety area for the storage and 20% for the shuffle). The unroll space takes 20% of the storage space by default. The capacity of these three spaces of JVM heap can be set by `spark.storage.unrollFraction`, `spark.storage.memoryFraction`, and `spark.shuffle.memoryFraction`. For example, in our testbed cluster, we can set the `spark.executor.memory` as 2.6 GB of the 4 GB memory of the worker node, which means that the JVM heap size is set to a maximum of 2.6 GB. Then, the actual capacities of storage space and shuffle space are $2.6 \text{ GB} \times 0.9 \times 0.6 = 1.4 \text{ GB}$ and $2.6 \text{ GB} \times 0.9 \times 0.2 = 0.46 \text{ GB}$, respectively. Accordingly, the unroll space takes $1.4 \text{ GB} \times 0.2 = 0.28 \text{ GB}$.

3.3. RDD Caching Policy

The Spark platform offers diverse RDD caching options involving main memory and disks. Basically, the default option is the `MEMORY_ONLY`, where the RDD is maintained in the storage space described in Section 3.2 as a non-serialized Java object. If this storage space is insufficient for holding all RDDs, some of them would be evicted from the main memory based on a pre-defined cache replacement policy. However, whenever a non-cached RDD is required for task processing, this RDD should be re-created based on the *lineage* information which can result in substantial performance degradation in this `MEMORY_ONLY` caching policy.

Besides the `MEMORY_ONLY` option, Spark provides alternative `MEMORY_AND_DISK`, `DISK_ONLY`, and `OFF_HEAP` options. The `MEMORY_AND_DISK` option stores RDDs in the non-volatile disk when the storage space is not enough to store all required RDDs. Disks can consist of HDDs or SSDs; however, normal spindle disks have relatively poor read/write throughput, so that the overall execution time can be longer than that of the `MEMORY_ONLY` caching option. To address this problem, we can effectively leverage SSDs, which can potentially reduce the overall job completion time compared to the normal HDD-based approach.

The `DISK_ONLY` option stores RDDs only in non-volatile storage devices such as HDDs or SSDs, i.e., not in the main memory. A cluster which does not have sufficient amount of available memory can achieve good performance with this option. In this case, since RDD is stored only in disk media, the shuffle space can be extended instead of using the storage space of the memory. As a result, when running an application such as PageRank, which generates a relatively large amount of shuffle data, we can observe better performance than in the `MEMORY_ONLY` case.

The `OFF_HEAP` option enables Spark to use off-heap space, which is outside of the Java garbage collector's management. Thus, if we use off-heap space, we have to deal with complicated memory operations such as allocation/deallocation and serialization/deserialization. Therefore, for practical purpose, we do not use the `OFF_HEAP` configuration.

3.4. Optimization Methodology

As we discussed in Sections 3.2 and 3.3, our optimization methods include (1) the configuration of the Spark JVM heap and (2) the experimental options of the RDD caching policy as follows:

1. **Spark JVM heap configuration:** We investigated the effects of changing the capacity fraction ratios of the shuffle and storage spaces. The ratio of shuffle and storage space is 60%:30%, 50%:40%, and 20%:60%, respectively. The “20%:60%” of shuffle and storage ratio is the default value in the Spark setup. We choose “60%:30%” to contrast the result with sufficient shuffle space and configure “50%:40%” to show the performance in a balanced way.
2. **RDD caching policy:** We also examined the effects of different RDD caching policies. We compared the performance of various policies such as OFF_HEAP, MEMORY_ONLY, MEMORY_AND_DISK, and DISK_ONLY, where DISK denotes the SSD in this experiment.

Table 3 shows total 12 different experimental configurations based on the RDD caching policies and Spark JVM capacity fraction ratios. In the experiment configurations labeled with “_1” (for example, “N_1”), we set 60% of the Spark JVM heap for shuffling and 30% for storage spaces. With those labeled “_2”, we set 50% of the Spark JVM heap for shuffling and 40% for storage. Finally, for those labeled with “_3”, we set 20% of the Spark JVM heap for shuffling and 60% for storage, as can be seen from the “Option”, “Shuffle”, and “Storage” columns in Table 3. Note that our testbed cluster’s maximum memory size of the executor is 2.7 GB, i.e., each worker node has 2.7 GB as the Spark JVM heap size.

Table 3. Experimental configurations: Spark JVM heap and RDD caching policy.

Option	Caching Policy	Shuffle	Storage
N_1	No Caching	60%: 1.62 GB	30%: 0.81 GB
N_2	No Caching	50%: 1.35 GB	40%: 1.08 GB
N_3	No Caching	20%: 0.54 GB	60%: 1.62 GB
M_1	Memory only	60%: 1.62 GB	30%: 0.81 GB
M_2	Memory only	50%: 1.35 GB	40%: 1.08 GB
M_3	Memory only	20%: 0.54 GB	60%: 1.62 GB
M&S_1	Memory and SSD	60%: 1.62 GB	30%: 0.81 GB
M&S_2	Memory and SSD	50%: 1.35 GB	40%: 1.08 GB
M&S_3	Memory and SSD	20%: 0.54 GB	60%: 1.62 GB
S_1	SSD only	60%: 1.62 GB	30%: 0.81 GB
S_2	SSD only	50%: 1.35 GB	40%: 1.08 GB
S_3	SSD only	20%: 0.54 GB	60%: 1.62 GB

In terms of the RDD caching policy, the “N” option is not to cache the RDD, “M” option is to cache the RDD on the memory only, “M&S” option is to cache the RDD on the memory and SSD together, and finally, “S” option is for caching the RDD on the SSD only.

Through our experiments, we propose optimizing strategies that can achieve the best performance from the cluster which has insufficient memory amounts by carefully adjusting the Spark JVM heap configuration and employing an effective RDD caching policy, as we will see in Section 4.

4. Experimental Results and Analysis

4.1. 500 MB PageRank Experiments

4.1.1. Results with Changing JVM Heap Configurations

Figure 3 shows the experimental results of each stage in the PageRank workload by changing the JVM heap sizes. In the *Distinct* stage, Spark reads the input data and distinguishes the URL and links. As we can see from the results of the *Distinct0* stage,

the overall execution time decreases by changing the JVM heap sizes from _1 and _2 to _3 options, mainly due to the *garbage collection* (GC). For example, the GC time takes 25 s, 24 s, and 16 s in M&S_1, M&S_2, and M&S_3, respectively. Therefore, in the *Distinct0* stage, as we increase the amount of storage space, we can improve the overall performance by reducing the GC time. On the other hand, in the *Distinct1* stage, the overall execution time increases as we change the options from _1 and _2 to _3. This is mainly because of the *shuffle spill*. When we checked the Spark web UI, the shuffle data were spilled onto the disk because of the lack of shuffle memory space. For example, the sizes of shuffle spill data on disk in M&S_1, M&S_2, and M&S_3 are 0, 220 MB, and 376 MB respectively. When the shuffle spill occurs, the CPU overheads for spilling the data onto the disk increase because the data need to be serialized.

Figure 3. The PageRank job execution time for 500 MB dataset per stage (s). Each shade represents each stage in a Spark job. Overall, M&S_1 case shows the best performance.

After the *Distinct* stages, there are *iterative flatMap* stages to obtain ranks. Basically, *flatMap* stages generate a lot of shuffle data, which can make our cluster lack the necessary shuffle memory space. Therefore, as the available amount of shuffle space decreases (in order from options _1, _2, and _3), the more shuffle spill can occur, which can potentially affect the overall job execution time (e.g., M&S option *flatMap2* stage _1: 37 s, _2: 40 s, _3: 49 s). However, when the data are cached on memory only (i.e., M_1, M_2, and M_3), they show another pattern. The main reason for this behavior is that the Spark scheduler schedules the tasks unevenly because there is a lack of memory storage space for caching the RDD on options _1 and _2. If a worker does not have RDDs, it is excluded from the scheduling pool. Therefore, the other workers have to handle additional tasks with GC overheads which can affect the whole job execution time.

4.1.2. Results with Changing RDD Caching Options

First of all, *Distinct* stages are not affected by changing the RDD caching policy but only by the memory usage. The stages that are affected by the RDD caching option are *flatMap* stages since during the shuffle phase, cached RDDs are used again.

In Figure 4, the graph is normalized by the N_1 option that does not cache the RDD and _1 memory configuration to check the performance difference. When comparing only the graphs of _1, in the order of M_1, M&S_1, and S_1, there is a 32% performance degradation in M_1 and 30% and 20% performance improvements with M&S_1 and S_1, respectively. With the M_1 option, the reason for relatively poor performance is that RDDs are cached unevenly due to the lack of storage, which will result in uneven scheduling as we previously mentioned. This means that the JVM heap space is insufficient for shuffling the data and saving the RDDs.

Figure 4. The normalized PageRank job execution time for 500 MB dataset.

In order to address this problem, we spread the RDDs to cache both on the memory and SSD, which can actually improve the performance as shown with the M&S_1 option. Caching the RDD on memory improves the access speed for the RDD, and caching the RDD on the SSD can avoid the shuffle spill by effectively extending the available shuffle space in memory. With the S_1 option that has shown a 20% performance improvement, the RDD is cached on the SSD only. The shuffle spill is reduced by caching the RDD on the SSD. However, it has achieved a lower performance improvement than M&S_1, where the RDD is mainly cached on memory and reused from the memory.

In the default configuration of Spark, which is option _3, we can see that in the order of M_3, M&S_3, S_3, and N_3, overall performance decreases. In the default configuration, the storage of the JVM heap is enough for the RDD to be cached with balance. Therefore, the overall performance mainly depends on the performance of the memory device used. However, we can still see the best performance with the M&S_1 option since we can effectively reduce the GC time and shuffle spill by caching the RDD *both* on the memory and SSD.

4.2. 1 GB PageRank Performance

We experimented with the PageRank workload by increasing the size of data from 500 MB to 1 GB. Figure 5 shows the different behaviors of the system compared to PageRank for the 500 MB dataset. We can see some failed jobs that could not succeed in completing the job until the *take6* stage (e.g., N_1, N_2, M_1, M_2, M_3, M&S_3). Among these failed

jobs, there are ones that failed in *flatMap2* stage, which are N_1, N_2, and M_1. The reason for the job failure is the lack of storage memory. The GC occurs when the RDD is cached on insufficient memory. Because of this GC overhead, the Spark executor receives an *ExecutorLostFailure* exception.

M_2, M_3, and M&S_3 could proceed with the processing until the *flatMap2* stage; however, after this, failure occurs. M&S_3 operates similarly to the M_3 until the *flatMap2* stage because when M&S_3 option is used, there is sufficient memory to cache the RDD. After the *flatMap2*, the *OutOfMemory* error occurs due to the lack of shuffle memory space in the *flatMap3* stage.

4.2.1. Results with Changing JVM Heap Configuration

The *Distinct0* stage basically shows very similar results with the 500 MB dataset, and the overall performance improves in the order of options _1, _2, and _3. This is because the GC time is reduced to 78 s, 59 s, and 28 s, respectively.

On the other hand, in the *Distinct1* stage, it showed different results for the 500 MB dataset. In the 500 MB dataset experiment, we can see the performance gain by increasing the shuffle space of memory. However, in the 1GB dataset experiment, the executor memory of the worker node cannot accommodate the large size of data. Therefore, the shuffle space of memory becomes relatively insufficient. For example, the amounts of shuffle spill for options _1, _2, and _3 are 575.5 MB, 813.8 MB, and 843.4 MB, respectively, and the GC time takes 33 s, 10 s, and 8 s, respectively. As we mentioned before, when shuffle spill occurs, the RDD needs to be serialized so that the CPU computations can increase, which can result in the overall performance degradation.

Figure 5. The PageRank job execution time for 1 GB dataset(s).

4.2.2. Results with Changing RDD Caching Policy

For analyzing execution time with changing the RDD caching policy, as we can see from Figure 6, we exclude distinct stages from Figure 5. This is because we do not need to analyze *distinct* stages since there are no changes caused by changing the RDD caching policy.

Interestingly, there are no changes with various JVM heap configurations in *flatMap* stages as opposed to the case of the 500 MB dataset. The reason for this is that the shuffle spill occurs in all configurations because there is insufficient memory. The overall execution time by changing the RDD caching option increases in the order of M&S, S, N, and M. (M&S is the fastest option.) In option N, the *ExecutorLostFailure* error occurs because there is no sufficient memory space. In option M, when the RDD is cached on memory, GC overhead occurs because there is insufficient memory space. Even if the RDD is cached on memory, the job fails because of the *ExecutorLostFailure* error that occurs when the shuffle memory space is insufficient (*OutOfMemory*).

In such low available memory situations, M&S and S options can be effective alternatives. In the M&S_1 option, we increase the accessibility of the RDD by caching the RDD using both the memory and SSD. As a result, there is a performance improvement for the same reason as the 500 MB dataset. In addition, there is sufficient shuffle memory space due to caching the RDD on the SSD. As seen in Figure 6, the M&S_1 option becomes the fastest option in this experiment (M&S_1:0.6, S_1:0.63, T_1 0.64).

Figure 6. The normalized PageRank job execution time for 1 GB dataset.

4.3. TC Experiment Analysis

Figure 7 shows the results of TC (transitive closure) experiments that use the input data involving 50,000 edges and 25,000 vertexes generated at random. The iteration number is 10. Through the iterations, the number of tasks are doubled at each iteration, and therefore, the size of the RDD increases and the amounts of shuffle read and write also increase. In the last iteration, the number of tasks becomes 4096. As there are more iteration stages, there is a larger effect on the total job execution time, and the last iteration stage is the biggest stage, consisting of many tasks that can lower the overall performance.

Figure 7. The normalized TC job execution time.

As we can see from Figure 7, the performance improves in the order of _3, _2, and _1 on the M, M&S, and S options, which means that obtaining a sufficient shuffle memory of the JVM heap is helpful. With the M option, the performance of option _1 is 18% faster than option _3, whereas in the M&S option, the performance of option _1 is 3% faster than _3. In the S option, the performance of _1 is 2% faster than _3.

When we focus on changing the RDD caching option, the performance of option S_1 is 42% faster than N_1, and it is also 31% faster than M_1. The reason for the performance gain of job execution time is dependent on the last iteration stage. The key factor affecting the last iteration stage is the shuffle read blocked time. Shuffle read blocked time occurs when the RDD executed in the previous stage is read from another worker node through the network because of the lack of executor memory. Even if each task has a performance gain of about 1–2 s through solving the shuffle read blocked time, we can achieve a significant performance gain because in the last state, the number of tasks is quite large (i.e., 4096). In addition, one of the main factors affecting job execution time is the count stage, which counts how many edges the TC matrix has at the last job. With option N, because there are no RDDs cached in the count stage, Spark reads the shuffle data executed from previous stage, which takes 60 s. In addition, in option M, the RDD is not cached on memory because of the lack of the executor memory. As a result, it also takes 60 s. However, in the M&S and S options, the RDD can be cached on the memory and SSD, so that it takes only 2 s in the count stage.

4.4. TeraSort Experiment Analysis

Figure 8 shows the experimental results of the TeraSort benchmark that uses a 10 GB dataset by changing the JVM heap configuration and RDD caching option. This graph is normalized by option N_1. We can see that all of the job execution times are similar; the difference between them is less than 5%. In the TeraSort workload, there were no performance improvements or degradations by changing the configurations and options. In the sorting stage, there are a few shuffles through the network. However, the sizes of shuffle read and shuffle write are 25 MB each, which is quite small compared to PageRank and TC. Therefore, the JVM heap configuration and RDD caching option do not affect the performance. Furthermore, the TeraSort workload does not consist of iterative jobs as in the transitive closure, so that there is no benefit from RDD caching in the previous stage.

Figure 8. The normalized TeraSort job execution time for 10 GB dataset.

4.5. K-Means Clustering Experiment Analysis

The normalized job completion time of k-means clustering for the 1.5 GB dataset is shown in Figure 9. The purpose of the k-means clustering is finding the k clusters in the dataset based on the distance measurement (e.g., Euclidean distance). In this workload, the algorithm reduces the SSE (sum of squared error) [24] through iterating the distance calculation between the k center points and each data point. In this experiment, we iterate this process eight times. The amount of data to be shuffled is very small because the data needed from the previous stage are the information about the center points and SSE in each stage. In our k-means clustering workload, the maximum amount of shuffle read/write data is 1.0 MB, and the minimum is 0.8 MB. Here, the shuffle spill does not occur because the shuffle space is sufficient in all settings. In the experiments with no caching options, there is no difference among options _1, _2, and _3, because these settings do not cache any RDD, and in all three setting, the shuffle space is sufficient.

Figure 9. The normalized k-means clustering job execution time for 1.5 GB dataset.

When caching the RDDs in the main memory or the memory and SSD, the more the storage space for the RDD, the more the performance in the job execution time improves because more RDDs can be cached on the storage space. When comparing the *memory_only* option and *memory_and_SSD* option, the *memory_and_SSD* option showed the better performance improvement. This is because in the *memory_only* option, the storage space is insufficient even in the *M_3* option. In addition, caching the RDDs on the SSD solves such a lack of storage memory. *Memory_and_SSD* options improved performance by 10% on average as compared with the *memory_only* option.

Note that the k-means clustering workload shows an opposite performance tendency from PageRank and transitive closure workloads because of the difference in the amount of shuffle data. We will discuss this in more detail in the next subsection.

5. Discussion and Summary

5.1. Discussion

We analyzed the main factors of potential performance degradation problems based on the feature of workload and processing stages. Our extensive experimental results are summarized with respect to applying the performance optimization techniques of the Spark platform for various workloads as follows:

- The performance degradation by Java garbage collection:** In the PageRank workload with the 500 MB dataset and 1GB dataset, the GC occurs when there is insufficient storage space of the JVM heap to store the RDD. In the *Distinct0* stage that reads the input file from the HDFS and caches it into the RDD, the GC occurs. We expand the storage space of the JVM heap through the configuration to solve this GC problem. We can improve performance to reduce GC because the storage space of the JVM heap can be expanded. In Figures 3 and 5, with the same RDD caching option, the *_3* configuration shows the best performance in the *Distinct0* stage. In addition, in PageRank with the 1 GB dataset, some options fail in the *flatMap* stage due to the lack of memory. The GC overhead increases so much that the stage fails or goes into an infinite loop. Thus, we construct the cluster with SSDs to solve this problem. It shows a performance improvement and succeeds in the job that failed using only memory, as seen in Figure 6, *M&S_1* and *S_1*.
- The performance degradation by shuffle spill:** In the PageRank workload with the 500 MB dataset and 1 GB dataset, in the *flatMap* stage, we can see that option *M&S_1* shows the best performance because it has the least amount of shuffle spill (Figure 4: *M&S_1* is 30% faster than *N_1*; Figure 6: *M&S_1* is 40% faster than *N_3*). PageRank has many shuffle tasks. Thus, when the shuffle space of the JVM heap is insufficient for shuffling the data through the network, the shuffle spill occurs. Therefore, in order to reduce the shuffle spill, expanding the shuffle space of the JVM heap becomes the key factor of performance improvement. Furthermore, we can improve the performance by storing the RDD both on the memory and SSD. This can make the executor expand the shuffle memory of the JVM heap to reduce shuffle spill. If there are more iterations, the performance from the *flatMap* stage would be the key point of the performance improvement. In the 1GB dataset experiment, the job execution time of *S_3* is the best option, because RDDs are cached on the SSD only, and there is sufficient heap memory on the executors. Thus, in the *S_3* option, the *Distinct* stages are faster than any other option. However, if the iteration number increases, the *flatMap* stage affects the job execution time. Thus, the *M&S_1* option can achieve great performance in this case. Through these analyses, we can identify that shuffle has a key effect on job completion time. Thus, we have to expand the shuffle memory of the JVM heap and cache the RDD both in the memory and SSD to obtain enough shuffle memory space for preventing shuffle spill.
- The performance degradation by shuffle read blocked time:** There is shuffle read blocked time on the TC workload. It occurs when there are a lot of tasks in the stage

and each task needs to read the previous RDD through the network. As the result of the TC experiment in Figure 7, the M&S option is faster than option M. In the same RDD caching option, expanding the shuffle space of the JVM heap is faster than expanding the storage space. The reason for improved performance is that by expanding shuffle space of the JVM heap, the shuffle read blocked time decreases in each task.

5.2. Summary: Which Is the Best Way?

In the comprehensive experimental results, there is not a single best setup to boost all workloads, since each of these workloads has different characteristics, even in its job lifetime. However, we can still propose how to optimize the configurations of a distributed in-memory computing platform by considering the variety of target workloads as follows:

- **Spark JVM heap configuration—shuffle area vs. storage area:** According to the experimental results of four different workloads, we can observe the performance differences depending on workload characteristics. For example, PageRank is a typical example of having a large amount of shuffle data so that allocating more memory to the shuffle portion improves the overall performance. However, in the case of k-means clustering, the more we allocate to storage memory, as opposed to shuffle memory, the less execution time is required. Therefore, if we can adjust the JVM memory allocation percentage dynamically according to the workload characteristics, we can optimize the total execution time. The Hadoop YARN [25] enables us to assign jobs to different types of clusters (configurations), so that we can apply this idea to a large sized Hadoop cluster to meet memory characteristics for various types of jobs.
- **RDD caching policy—memory vs. SSD:** In most cases, the SSD-backed memory caching shows the best performance unless all of the RDDs can fit within the actual main memory. Therefore, the SSD-assisted memory caching policy can be a viable choice for challenging workloads requiring substantial amounts of main memory that cannot be met by any single node in a cluster.

6. Conclusions

In this paper, we have investigated the main factors for the performance degradation of the Spark system running on top of a commodity-server-based computing cluster with insufficient available main memories. After experimentation and analysis, we presented alternatives that can improve the overall performance.

Java garbage collection occurs when the storage space of the JVM heap is insufficient due to a lack of physical memory. Java GC makes tasks wait for garbage collection so that the overall job completion time increases. The shuffle spill occurs when the shuffle space of the JVM heap is insufficient during the shuffle phase. Shuffle spill increases the CPU overhead to perform serialization for spilling intermediate shuffle data to the disk due to the lack of shuffle space. In the TC workload experiment, shuffle read blocked time makes the task wait for reading shuffle data through the network because of the lack of shuffle space. All of these factors can potentially increase the overall job completion time which can seriously affect the performance of the Spark system.

To address these problems, we construct a cluster with an SSD and cache the RDD both on the memory and SSD separately by utilizing the SSD to supplement the storage space of the memory. In addition, we adjust the JVM heap configuration for expanding the shuffle space. As a result, we could achieve a 30% performance improvement for the PageRank workload and a 42% performance improvement for the TC workload. We have identified that the shuffle spill can be a key factor of performance degradation and showed through experimentation that in workloads consisting of a number of iterations and shuffling, expanding the shuffle space can provide significant performance gains. In addition, we found that different memory usage patterns of jobs can affect the total execution time depending on the storage/shuffle memory percentage allocation in the JVM. According to the performance analysis of PageRank and k-means clustering, memory allocation in

the JVM that is well-tuned to the workload characteristics can significantly improve job completion time.

Integrating these findings into the Spark platform would be one of our future works. For example, if workloads can be characterized in terms of the amounts of shuffle data, an optimized configuration can be automatically applied to accelerate the processing of target workloads. Therefore, in heterogeneous server configurations, developing a workload memory usage-aware scheduling system can improve the overall performance of a Spark-based cluster.

Author Contributions: Conceptualization, J.L. (Jaehwan Lee); methodology, J.L. (Jaehwan Lee) and J.C.; software, J.C. and J.L. (Jaehyun Lee); validation, J.C., J.L. (Jaehyun Lee) and J.L. (Jaehwan Lee); investigation, J.L. (Jaehwan Lee) and J.-S.K.; resources, J.L. (Jaehwan Lee) and J.-S.K.; data curation, J.C. and J.L. (Jaehyun Lee); writing—original draft preparation, J.C. and J.L. (Jaehyun Lee); writing—review and editing, J.L. (Jaehwan Lee) and J.-S.K.; visualization, J.L. (Jaehyun Lee); supervision, J.L. (Jaehwan Lee) and J.-S.K.; project administration, J.L. (Jaehwan Lee) and J.-S.K.; funding acquisition, J.L. (Jaehwan Lee). All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by the Basic Science Research Program (NRF-2020R1F1A1072696) through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT, GRRC program of Gyeonggi Province (No. GRRC-KAU-2017-B01, “Study on the Video and Space Convergence Platform for 360VR Services”), and ITRC (Information Technology Research Center) support program (IITP-2021-2018-0-01423).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Available upon request.

Conflicts of Interest: The authors declare no conflict of interest.

References

- Dean, J.; Ghemawat, S. MapReduce: Simplified data processing on large clusters. *Commun. ACM* **2008**, *51*, 107–113. [CrossRef]
- The Apache Hadoop Project: Open-Source Software for Reliable, Scalable, Distributed Computing. Available online: <https://hadoop.apache.org/> (accessed on 10 September 2021).
- Shvachko, K.; Kuang, H.; Radia, S.; Chansler, R. The hadoop distributed file system. In Proceedings of the 2010 IEEE 26th symposium on mass storage systems and technologies (MSST), Incline Village, NV, USA, 3–7 May 2010; pp. 1–10.
- Zaharia, M.; Chowdhury, M.; Franklin, M.J.; Shenker, S.; Stoica, I. Spark: Cluster computing with working sets. *HotCloud* **2010**, *10*, 95.
- Ousterhout, K.; Rasti, R.; Ratnasamy, S.; Shenker, S.; Chun, B.G. Making sense of performance in data analytics frameworks. In Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI), Oakland, CA, USA, 4–6 May 2015; pp. 293–307.
- Xing, W.; Ghorbani, A. Weighted pagerank algorithm. In Proceedings of the IEEE Second Annual Conference on Communication Networks and Services Research, Fredericton, NB, Canada, 21 May 2004; pp. 305–314.
- Chakradhar, S.T.; Agrawal, V.D.; Rothweiler, S.G. A transitive closure algorithm for test generation. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* **1993**, *12*, 1015–1028. [CrossRef]
- O’Malley, O. Terabyte Sort on Apache Hadoop. Yahoo. May 2008. pp. 1–3. Available online: <http://sortbenchmark.org/YahooHadoop.pdf> (accessed on 10 September 2021).
- K-Means Clustering. Available online: https://en.wikipedia.org/wiki/K-means_clustering (accessed on 10 September 2021).
- Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J.; McCauley, M.; Franklin, M.J.; Shenker, S.; Stoica, I. 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), San Jose, CA, USA, 25–27 April 2012; pp. 15–28.
- Davidson, A.; Or, A. *Optimizing Shuffle Performance in Spark*; Technical Report; Berkeley-Department of Electrical Engineering and Computer Sciences, University of California: Berkeley, CA, USA, 2013.
- Nicolae, B.; Costa, C.H.A.; Misale, C.; Katrinis, K.; Park, Y. Leveraging Adaptive I/O to Optimize Collective Data Shuffling Patterns for Big Data Analytics. *IEEE Trans. Parallel Distrib. Syst.* **2017**, *28*, 1663–1674. [CrossRef]
- Zhang, H.; Cho, B.; Seyfe, E.; Ching, A.; Freedman, M.J. Riffle: Optimized Shuffle Service for Large-Scale Data Analytics. In *Proceedings of the Thirteenth EuroSys Conference*; EuroSys ’18; Association for Computing Machinery: New York, NY, USA, 2018. [CrossRef]

14. Pu, Q.; Venkataraman, S.; Stoica, I. Shuffling, Fast and Slow: Scalable Analytics on Serverless Infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*; USENIX Association: Boston, MA, USA, 2019; pp. 193–206.
15. Zhou, P.; Ruan, Z.; Fang, Z.; Shand, M.; Roazen, D.; Cong, J. Doppio: I/O-Aware Performance Analysis, Modeling and Optimization for In-memory Computing Framework. In *Proceedings of the 2018 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Belfast, UK, 2–4 April 2018; pp. 22–32.
16. Marcu, O.C.; Costan, A.; Antoniu, G.; Pérez-Hernández, M.S. Spark Versus Flink: Understanding Performance in Big Data Analytics Frameworks. In *Proceedings of the 2016 IEEE International Conference on Cluster Computing (CLUSTER)*, Taipei, Taiwan, 13–15 September 2016; pp. 433–442. [[CrossRef](#)]
17. Petridis, P.; Gounaris, A.; Torres, J. Spark Parameter Tuning via Trial-and-Error. In *Advances in Big Data*; Angelov, P., Manolopoulos, Y., Iliadis, L., Roy, A., Vellasco, M., Eds.; Springer International Publishing: Cham, Switzerland, 2017; pp. 226–237.
18. Gounaris, A.; Torres, J. A Methodology for Spark Parameter Tuning. *Big Data Res.* **2018**, *11*, 22–32. [[CrossRef](#)]
19. Yu, Z.; Bei, Z.; Qian, X. Datasize-Aware High Dimensional Configurations Auto-Tuning of In-Memory Cluster Computing. *SIGPLAN Not.* **2018**, *53*, 564–577. [[CrossRef](#)]
20. Veiga, J.; Expósito, R.R.; Taboada, G.L.; Touriño, J. Enhancing in-memory efficiency for MapReduce-based data processing. *J. Parallel Distrib. Comput.* **2018**, *120*, 323–338. [[CrossRef](#)]
21. Lion, D.; Chiu, A.; Sun, H.; Zhuang, X.; Grcevski, N.; Yuan, D. Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*; USENIX Association: Savannah, GA, USA, 2016; pp. 383–400.
22. Maas, M.; Harris, T.; Asanović, K.; Kubiawicz, J. Trash Day: Coordinating Garbage Collection in Distributed Systems. In *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*; USENIX Association: Kartause Ittingen, Switzerland, 2015.
23. Wang, B.; Tang, J.; Zhang, R.; Ding, W.; Qi, D. LCRC: A Dependency-Aware Cache Management Policy for Spark. In *Proceedings of the 2018 IEEE International Conference on Parallel Distributed Processing with Applications, Ubiquitous Computing Communications, Big Data Cloud Computing, Social Computing Networking, Sustainable Computing Communications (ISPA/IUCC/BDCloud/SocialCom/SustainCom)*, Melbourne, VIC, Australia, 11–13 December 2018; pp. 956–963.
24. Error Sum of Squares (SSE). Available online: https://hlab.stanford.edu/brian/error_sum_of_squares.html (accessed on 10 September 2021).
25. Vavilapalli, V.K.; Murthy, A.C.; Douglas, C.; Agarwal, S.; Konar, M.; Evans, R.; Graves, T.; Lowe, J.; Shah, H.; Seth, S.; et al. Apache Hadoop YARN: Yet Another Resource Negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC'13)*, Santa Clara, CA, USA, 1–3 October 2013.