



A Task-Aware Fine-Grained Storage Selection Mechanism for In-Memory Big Data Computing Frameworks

Bo Wang¹ · Jie Tang² · Rui Zhang³ · Jialei Liu¹ · Shaoshan Liu⁴ · Deyu Qi²

Received: 26 March 2019 / Accepted: 19 May 2020 / Published online: 5 June 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract

In-memory big data computing, widely used in hot areas such as deep learning and artificial intelligence, can meet the demands of ultra-low latency service and real-time data analysis. However, existing in-memory computing frameworks usually use memory in an aggressive way. Memory space is quickly exhausted and leads to great performance degradation or even task failure. On the other hand, the increasing volumes of raw data and intermediate data introduce huge memory demands, which further deteriorate the short of memory. To release the pressure on memory, those in-memory frameworks provide various storage schemes options for caching data, which determines where and how data is cached. But their storage scheme selection mechanisms are simple and insufficient, always manually set by users. Besides, those coarse-grained data storage mechanisms cannot satisfy memory access patterns of each computing unit which works on only part of the data. In this paper, we proposed a novel task-aware fine-grained storage scheme auto-selection mechanism. It automatically determines the storage scheme for caching each data block, which is the smallest unit during computing. The caching decision is made by considering the future tasks, real-time resource utilization, and storage costs, including block creation costs, I/O costs, and serialization costs under each storage scenario. The experiments show that our proposed mechanism, compared with the default storage setting, can offer great performance improvement, especially in memory-constrained circumstances it can be as much as 78%.

Keywords Big data · In-memory computing · Storage scheme · Performance optimization

✉ Jie Tang
cstangjie@scut.edu.cn

Extended author information available on the last page of the article

1 Introduction

In-memory computing directly caches data into memory for future reusing and gets rid of burdensome disk access. As a result, In-memory big data computing frameworks, compared with disk-based computing frameworks like MapReduce [1], Hadoop,¹ and Dryad [2], can provide orders of magnitude improvement in terms of response time and processing efficiency [3]. Spark,² which is one of such in-memory computing frameworks originating from AMP, provides an immutable distributed collection of objects named *Resilient Distributed Datasets (RDDs)* [4] for memory state reserving and data sharing across the jobs. As data sharing in memory is 10–100 times faster than network and disk, Spark can process data up to 100× faster than Hadoop on occasions, especially in iterative and interactive computing. Therefore, Spark has been extensively used in companies/organizations, such as Tencent, Alibaba, Databricks, Amazon, and so on, and takes care of core service like advertisement recommendation and graph data mining. We can see the big trend that big data computing is shifting from on-disk processing toward in-memory computing speedy [5].

However, memory capacity, for an in-memory computing framework, would be a serious effect on computing performance. If memory demands of a task can not be met, its execution will be severely delayed by frequent garbage collections, task re-computing, and so on. Studies have shown that memory has become a performance bottleneck in Spark [6, 7]. Despite the continuous dropping in RAM prices and the increasing availability of high RAM servers, memory cache remains a constrained resource in large clusters and memory-constrained systems are always ubiquitous [5]. On the other hand, data production is growing at a rate that doubles every 2 years, and makes the applications' data volume explosion. As a result, the gap between input data and memory capacity is widening. In execution, the system produces large amounts of intermediate data, which makes the problem further worse. For example, for PageRank (3 iterations) and ConnectedComponents from twitter-rv dataset, the peak size of their RDDs will be 155.8 GB and 357.1 GB, respectively, much larger than input (24.3 GB) [8]. Such overhead is attributed to memory consumption of Java meta-data storage and objects' internal pointers [9].

In order to alleviate memory pressure, serialization is a popular solution to reduce the memory size of an RDD. After serialization, the size of an RDD is to $0.2\times-0.5\times$ of the original object which is approximately the size of the raw data [9]. On the flip of a coin, it introduces no negligible computing overhead, spending about 5% execution time for operation serialization. Another solution is using flexible storage mechanisms. Data are not only cached in memory but also cached on disk. In Spark, the cache RDDs are specified by users and the scheme for an RDD is manually specified either in the configuration file, which is a global way for storage scheme settings, or codes. Although the above approach is simple and convenient, it has three drawbacks as follows.

¹ Apache Hadoop Project, <http://hadoop.apache.org/>.

² Apache Spark Project, <http://spark.apache.org/>.

First, it is difficult and time-consuming to obtain the optimal storage scheme for each cache data block. For a system with m kinds of storage schemes, the number of storage schemes option can be m^n for all n blocks of cache data. The optimum search cost is exponentially increased to amount of data blocks in the application. And, it may become complicates when there are multiple applications running simultaneously. Moreover, jobs are sensitive to storage schemes. The execution time for the same application with different schemes setting differs a lot. An inappropriate storage scheme would result in poor system performance even application failure. So, it requires the user has a wealth of knowledge of data access pattern to avoid inappropriate storage schemes.

Second, data storage scheme is static throughout the execution of the job. After setting, storage schemes of all data are fixed during the entire application running. A new storage scheme can take effect only after the user recompiles and restarts the application. Memory state, including free memory size and in memory data, etc., which determines the success of a job, is variable. The scheme should capture the real-time memory state and makes an adjustment in time. If not, the job may fail.

Third, systems-support storage scheme setting takes a uniform way that each block of one RDD will follow the same storage assignment. Such coarse grained setting ignores the disparity of running tasks. As a result, it may be expected to result in a straggler. For some systems, straggler is a major factor hindering the performance of the system.

In this paper, we present a novel storage scheme selection mechanism, Task-aware Fine-grained Storage scheme auto-selection (TFSS), which can automatically make a decision on a block storage scheme. With its cost-based selection policy, minimal execution cost storage scheme will be set for blocks in the application. Our approach provides benefits over existing storage scheme selection mechanism in four aspects:

First, TFSS selects storage scheme for a block automatically rather than manually.

Second, TFSS makes a decision on a cost model which can characterize real-time memory state.

Third, TFSS is fine-grained. Instead of assigning all blocks of the same RDD with the same storage scheme, TFSS makes the storage scheme decision for each block individually.

Fourth, TFSS is aware of future tasks assignment on an executor and makes decisions by considering both current and future executive information.

The remainder of this paper is organized as following. Section 2 presents some backgrounds of data dependency and storage scheme selection mechanism in in-memory big data computing frameworks. Motivation is discussed in Sect. 3. Section 4 shows two models used in our auto-selection mechanism. The overall implementation details are presented in Sect. 5. Extensive experimental results are reported in Sect. 6. In Sects. 7 and 8, we survey related works and conclude this paper respectively.

2 Background

Unless otherwise specified, we limit our discussion to Spark in the following sections. However, this discussion applies equally to other in-memory big data computing frameworks such as Tez [10], Storm,³ and Flink.⁴

2.1 Data Dependency

Typical big data computing applications, such as deep learning, artificial intelligence, and stream processing, often perform well-defined workflows. For some prevalent in-memory big data computing frameworks, they use *Directed Acyclic Graphs (DAGs)* to represent these workflows. Although rich semantics of underlying data access patterns, which imply plenty of cache storage opportunities, are determined by the DAG, they are not well utilized in cache storage selection mechanisms of these frameworks.

In Spark, DAG of an application consists of RDDs for fault-tolerant and in-memory parallel computing. RDDs are partitioned and stored across a group of nodes in the cluster through a distributed storage system (e.g., HDFS [11], Amazon S3⁵). Each node holds not all but a subset of the blocks, each represents an RDD partition. A task is responsible for computing one block of the RDD. Before running, it is first assigned to one of the preferred nodes according to the scheduling policy (such as delay scheduler [12] based on the locality of the block) by TaskScheduler. All preferred locations of a task can be attained through *preferLocation()* API. So it makes it possible to obtain tasks that would run on an executor in advance.

DAG of an application is divided into stages by DAGScheduler in advance, and each stage holds a piece of code executed by tasks in it. During this process, shuffle dependencies, a many-to-one relationship among RDDs, are cut across different stages by the way of depth-first search. And each stage only holds RDDs with narrow dependencies, a one-to-one relationship. So the data access pattern of each task can be gotten in advance.

2.2 Storage Scheme Selection Mechanism in Spark

Programmers write an application by defining RDDs and operations on them. If an RDD is selected to be cached, the programmer will explicitly specify the caching storage scheme using *persist()* API in the code. For convenience, Spark provides a default value of MEMORY_ONLY which means the RDD only be cached in memory. Under this scheme, the task will efficiently perform when storage memory is sufficient. On the contrary, it will run with poor performance as a result of frequent re-computing, garbage collection, and so on.

³ Apache Storm Project, <http://storm.apache.org/>.

⁴ Apache Flink Project, <http://flink.apache.org/>.

⁵ Amazon S3, <https://aws.amazon.com/s3/>.

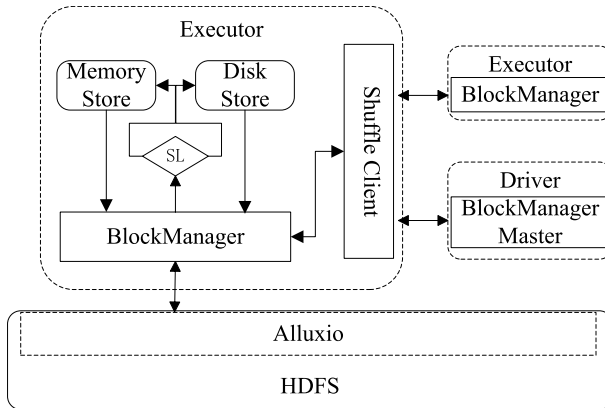


Fig. 1 Cache storage selection mechanism of Spark

As shown in Fig. 1, when computing a cache block, the *BlockManager*, lies on a worker, will firstly retrieve it on local or remote (if it not in local) from *BlockManagerMaster* which lies on the driver. If the block can be gotten, the *BlockManager* would return it. On the contrary, the *BlockManager* will cache it according to the storage scheme of the RDD to which it belongs. Except for memory, Spark also permits RDDs to be cached in other storage tiers, for example, disk. A block also can be serialized before it is cached. In practice, Spark provides some flags to indicate whether an RDD uses disk, memory, serialization. *BlockManager* stores the block into memory or disk through *MemoryStore* or *DiskStore* under indicating of these flags. During the process, Spark will evict some blocks within storage memory based on its replacement policy when there is no enough free memory to hold the block. By default, it uses *Least Recently Used (LRU)* policy. Although LRU can work effectively in most cases, more in-depth studies have shown that DAG-based cache replacement strategies, such as [5, 8], have better replacement patterns than LRU in Spark. Evicted blocks need to re-compute in the next access which may delay the task. Even worse, it may cause *Out Of Memory (OOM)* if there are no extra blocks to be evicted.

Further, during the execution of a block, Spark will perform different operations according to different situations (as shown in Fig. 2). In the initial state, a block is a sub-block after the RDD division. When it is accessed, Spark will recompute or directly read a block depending on the state of the block. The computed blocks are serialized or directly cached according to the storage scheme, where the cache operation can be cached on memory, disk, or both. Among them, the blocks cached to memory will be evicted from memory due to memory space. There are also two options for evicted blocks: to be discarded directly or stored on disk. Blocks that are directly discarded need to be recalculated when they are accessed again.

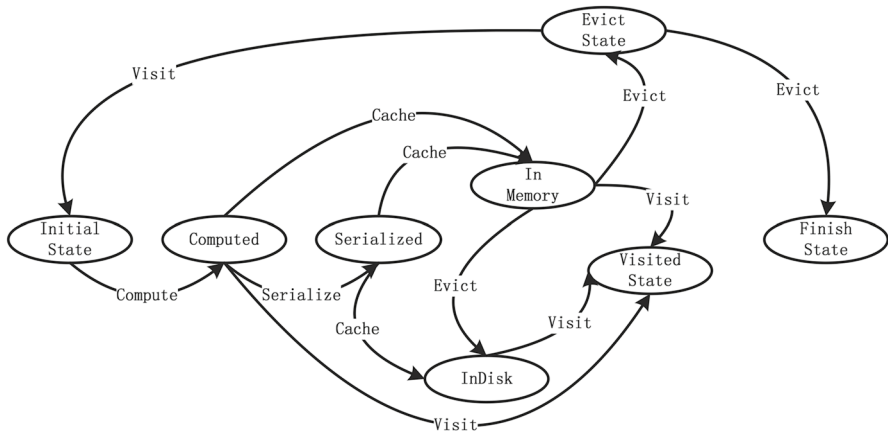


Fig. 2 State transition diagram for a block

Table 1 An overview of SparkBench suite

| Application type | Workload |
|-------------------|------------------------------|
| Machine learning | Logistic regression |
| | Linear regression |
| | Support vector machine (SVM) |
| | KMeans |
| | PCA |
| Graph computation | Page rank |
| | SVD Plus plus |
| | Triangle count |
| Other workloads | Connected component |
| | Decision tree |
| | Shortest paths |

3 Motivation

In this section, we illustrate the need for a novel storage selection mechanism in an in-memory big data computing framework from two perspectives. First, we compare the performance difference of the same benchmark under different storage schemes through experiments. Then, we also characterize the data access patterns in typical workloads through empirical studies. Experiments show that the workloads are sensitive to schemes, and a static storage scheme cannot capture the real-time memory state and make adjustments in time under a memory-constrained circumstance.

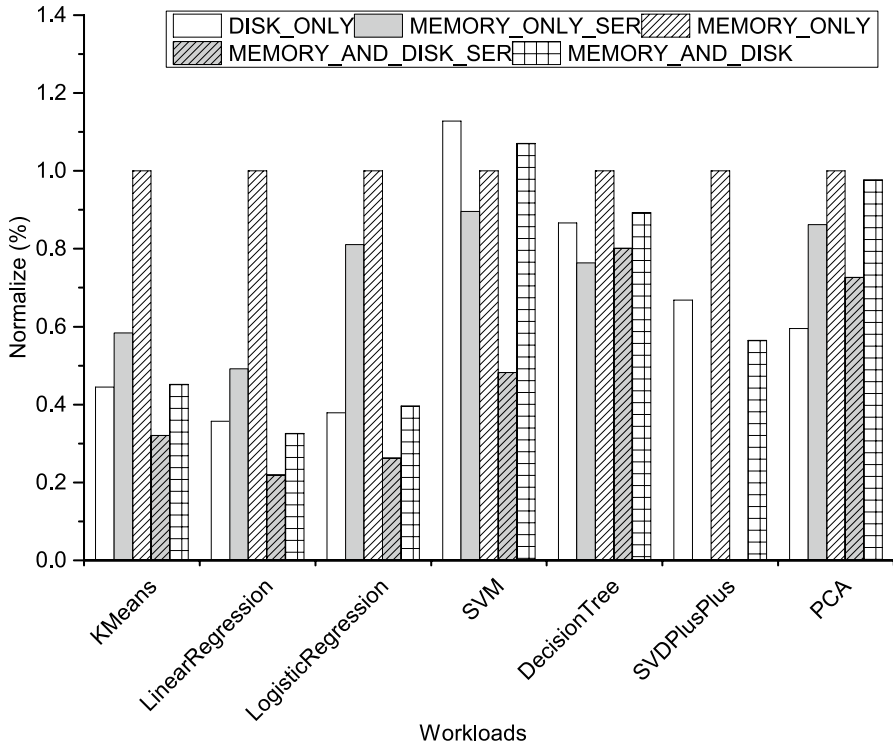


Fig. 3 Performance comparison of workloads in different storage schemes, execution time normalized by MEMORY_ONLY scheme

3.1 Methodology

We ran SparkBench [13], a popular workload suite, in a cluster of 4 heterogeneous instances (a master and three workers). We firstly measured the execution time under different cache storage schemes of 11 applications in SparkBench (as shown in Table 1), including machine learning, graph computation, etc. Furthermore, we characterized the data access patterns in executors through comparing their memory footprints.

3.2 Performance Comparison Under Different Storage Schemes

As described in the previous section, Spark combines one or more flags to form different schemes, such as MEMORY_AND_DISK scheme is a combination of disk flag and memory flag. As shown in Fig. 3, we compare the performance difference of 7 workloads in different storage schemes under a memory-constrained environment. And we draw two conclusions: First, there is a significant performance difference among different storage schemes. For LinearRegression, execution time in

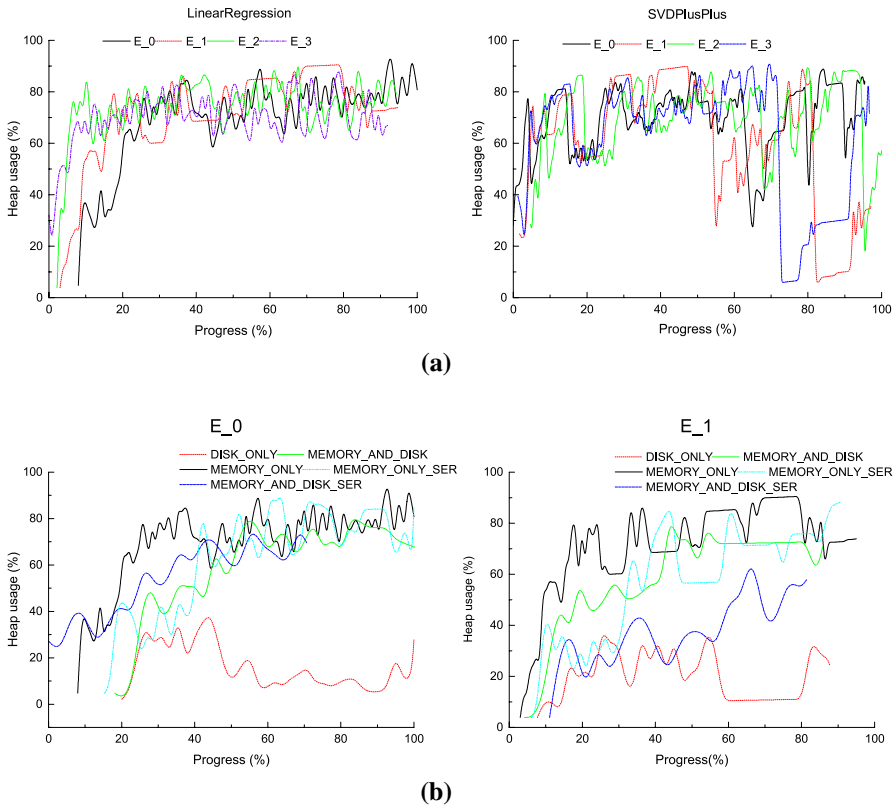


Fig. 4 **a** Heap usage of each executor in LinearRegression and SVDPlusPlus. **b** Heap usage of two executors with different storage schemes in LinearRegression

scheme MEMORY_ONLY is almost 450% (from 21.8 to 100%) of scheme MEMORY_AND_DISK_SER. SVDPlusPlus failed in both MEMORY_AND_DISK_SER and MEMORY_ONLY_SER two schemes but succeeded in other schemes. Second, the scheme MEMORY_ONLY, as default, is not always prominent in all schemes. For example, MEMORY_ONLY scheme has the longest execution time in KMeans, LogisticRegression, and LinearRegression.

3.3 Data Access Patterns

Figure 4 shows the memory footprints of all executors on a heterogeneous, memory-constrained cluster. And we can identify the following two common access patterns across the workloads:

1. *Memory footprints in all executors go peak quickly, then they become different from each other* Figure 4a shows memory footprints of LinearRegression and SVDPlusPlus two typical workloads in each executor. We find that memory foot-

print differs in each executor. For LinearRegression, which has the largest memory sensitivity in all workloads, the memory footprint of each executor reaches its peak soon after startup. And then, E_1 shows a different footprint with others for insufficient memory. In detail, the memory footprint of E_1 has two lasting peaks, while footprints of other executors still maintain a sharp wave shape. SVDPlusPlus, which crashes in multiple executors, shows the same characteristic. Different from LinearRegression, the memory footprint of a crash executor would reach peak again soon. The footprint of SVDPlusPlus shows that E_1 crashes in two locations of 52% and 81% respectively. Here, we consider a re-launch executor is the same as the previous executor.

2. *Memory footprints under different storage schemes have an apparent difference* As shown in Fig. 4b, the memory footprints of the same executor with different schemes have obvious differences in both heap usage and wave shape. The DISK_ONLY scheme has the lowest heap usage in all schemes. While the MEMORY_ONLY scheme has the fastest speed to get a peak in all schemes. Figure 4b further shows that the memory footprint of each executor is different not only in MEMORY_ONLY scheme but also in other schemes.

To summarize, we can draw three conclusions from the above experiments:

1. The workloads are sensitive to schemes, and the default scheme is not always the best one;
2. The framework has a greedy way to consume memory, and MEMORY_ONLY scheme has the fastest speed on consuming memory space in all schemes. When memory is insufficient, footprint becomes different in each executor for the delay causing by re-computation and garbage collection. So a static scheme would not catch the difference;
3. The memory footprints of the same workload under different schemes are different in all executors. So we can adjust the footprint of a workload through a flexible storage scheme.

Therefore, we propose a fine-grained task-aware storage scheme auto-selection mechanism to solve these problems.

4 Storage Cost Modeling

4.1 Cost Model

Before introducing the cost model of a block, we give some definitions in advance.

Cache Block A block, denoted as $b_{(i,j)}$, if it has been selected to be persisted for reusing, we call it a *Cache Block*. Here, i is the ID of the RDD it belongs to, and j is the index number of the RDD partition.

Cached Block A *Cache Block* $b_{(i,j)}$, if it has been resident in some storage tiers in access, we call it a *Cached Block*, denoted as $cb_{(i,j)}$.

Uncached Block A *Cache Block* $b_{(i,j)}$, if it is outside of all tiers of storage in access, we call it a *Uncached Block*, denoted as $ub_{(i,j)}$.

In access, a *Cached Block* must be kept in some storage tiers; it can be found in memory or disk. While an *Uncached Block* cannot be found in both memory and disk. It may not have been created yet or it has been evicted from storage. During the entire lifetime, a block can transform from one state into another under certain conditions. For example, a *Cached Block* will become an *Uncached Block* when it is discarded from memory. And an *Uncached Block* will become a *Cached Block* after being persisted into memory.

Execution Cost For a *Cache Block* $b_{(i,j)}$, its *Execution Cost* in its whole entire lifetime consists of two parts: *Creation Cost* and *Visiting Cost*. It can be defined as:

$$C_{(i,j)}^{execute} = m * C_{(i,j)}^{create} + n * C_{(i,j)}^{visit} \quad (1)$$

where m is the number of times that block $b_{(i,j)}$ is in *Uncached Block* state, and n is the number of times that it is in *Cached Block* state when it is visited. $C_{(i,j)}^{create}$ represents cost on creating the blocks in its longest path. And $C_{(i,j)}^{visit}$ represents cost on accessing the block when it is in cached.

Creation Cost For a *Cache Block* $b_{(i,j)}$, its *Creation Cost* is defined as the time spent on creating $b_{(i,j)}$. It can be defined as :

$$C_{(i,j)}^{create} = \sum_{b_{(i',j')} \in V} C_{(i',j')}^{compute} + C_{(i,j)}^{replace} \quad (2)$$

where V is a set of *Cache Blocks* which lie in longest path to create $b_{(i,j)}$ in the DAG. And $C_{(i',j')}^{compute}$ and $C_{(i,j)}^{replace}$ are defined in the following.

Computing Cost For a *Cache Block* $b_{(i,j)}$, its *Computing Cost*, denoted as $C_{(i,j)}^{compute}$, is defined as the time spent on computing $b_{(i,j)}$ under the circumstance that all its ancestors are *Cached Blocks*.

Replacement Cost For a *Cache Block* $b_{(i,j)}$, its *Replacement Cost* is defined as the time spent on replacing other blocks from memory to make room for it. The cost can be expressed as:

$$C_{(i,j)}^{replace} = \sum_{b_{(i',j')} \in \Omega} C_{(i',j')}^{evict} \quad (3)$$

where Ω is a set of *Cached Blocks* which are dropped for making room for $b_{(i,j)}$. Ω is a function of the replacement policy, as well as the free memory space in time. For example, it needs more blocks to be evicted from memory when less memory available.

Eviction Cost For a *Cached Block* $bc_{(i,j)}$, its *Eviction Cost*, denoted as $C_{(i,j)}^{evict}$, is defined as the time spent on dropping it from memory and/or persisting it on other storage tiers.

A block would be evicted in two cases. In the first case, a block is un-persisted and is selected by the arbiter for making room for coming *Cache Blocks*. Programmers can simply use the API `unpersist()` to specify *BlockManager* to drop it from

memory. In the second case, a block would be simply dropped or be further persisted to disk according to its storage scheme. For simplicity, we assume that the cost of dropping a block from memory is zero. Moreover, disk persistence needs to write the data into the disk and serialize data before writing (if needed), thus the cost here consists of disk I/O cost and serialization cost.

Visiting Cost For a *Cached Block* $bc_{(i,j)}$, its *Visiting Cost* is defined as the time spent on visiting the block. It can be expressed as:

$$C_{(i,j)}^{visit} = \alpha * \beta * C_{(i,j)}^{mem} + (1 - \alpha) * C_{(i,j)}^{disk} + \delta * C_{(i,j)}^{ser} \quad (4)$$

where $C_{(i,j)}^{mem}$ is defined as *Memory Cost*, and $C_{(i,j)}^{disk}$ is defined as *Disk Cost*. They are I/O costs for visiting $bc_{(i,j)}$ if it locates in memory and on disk correspondingly. $C_{(i,j)}^{ser}$, defined as *Serialization Cost*, is the cost on serializing $bc_{(i,j)}$. Here, α is the flag indicating whether the block is in memory or not. When a task visits a block, it first searches the block in memory. If the block is found, the task would read it directly from memory. Otherwise, the task would read the block from disk (if the block is in the disk). So *Memory Cost* and *Disk Cost* are exclusive in (4). β indicates serialization rate of $b_{(i,j)}$. As described in [9], the serialization rates of different datasets are not identical. The range of them is among [0.20–0.50]. δ indicates whether the block needs to be serialized or not. δ will further affect the value of β . For example, if δ is 0, β would be 1; otherwise if δ is 1, β is no more than 1.

For a *Cached Block*, its *Disk Cost* is 1–2 orders of magnitude higher than *Memory Cost*. Besides those two access costs are directly proportional to the size of the block, so *Serialization Cost* and *Computing Cost* are. Additionally, *Serialization Cost* of a block is relevant to its RDD type [8]. For simplification, we use the cost of per MB data (denoted as CPM) and block size to calculate the corresponding cost like *Disk Cost*, *Memory Cost* and *Serialization Cost*. The cost is equal to the product of CPM and block size. Moreover, Spark supports different serializers (such as JavaSerializer and KryoSerializer⁶). In default, the framework uses the same serializer in the whole program running cycle. Additionally, Zhao [14] proposes a serialization mechanism that provides different serializers for different blocks to improve storage performance. Although serialization rate and *Serialization Cost* of a block are different when using different serializers, it can be constant for the same serializer can be captured during task running. In our approach, we use the static serialization mechanism that Spark provides.

When a block has been cached, we can monitor the start time and end time then calculate those costs listed above: *Computing Cost*, *Disk Cost*, *Memory Cost* and *Serialization Cost*. If a *Cache Block* has not been cached, we can speculate its costs according to other blocks whose costs have been produced. As shown above, the costs of a block are proportion to its block size. CPM with the same RDD id can be the target blocks prefer to. If CPM with the same RDD id is absent, CPMs with other RDD id can be the candidate. If there are multi-CPMs, we use the CPM with the least computation. Additionally, the max, min, or mean value of the CPMs also

⁶ Kryo, <https://github.com/EsotericSoftware/kryo/>.

can be used for reference. Further, we update the corresponding CPMs of an RDD after an operation completed. For serialization operation, we serialize a block of each RDD for the first time that it is cached. After serialization, the serialization rate and *Serialization Cost* per MB of each RDD are learned. Additionally, serialization and de-serialization are counterpart operations. To simplify the computation, we assume that serializing and de-serializing of the same block has an identical cost.

As shown in Sect. 2, Spark supports to cache a block in memory, or on disk, or serialization. We further discuss *Visiting Cost* of a block in the following four scenarios.

4.1.1 Memory Scenario

In this scenario, with the default setting, a block is cached in memory in object form. The corresponding storage scheme is `MEMORY_ONLY`. Although the memory space occupied by the block in this scenario is larger than in other scenarios, the computing access speed of the block is fastest when the memory space is sufficient. Here, the cost of revisiting a block mainly comes from memory I/O. This serves as the original motivation for in-memory computing framework design. On the contrary, when revisiting, some blocks may absent in memory since being replaced by other blocks. At this time, the cost of revisiting a block mainly comes from re-computing. The number of blocks that need to be re-computed is inversely proportional to the size of available memory. In the worst case, it may cause task failure rooting from *OOM*.

4.1.2 Disk Scenario

In this scenario, a block is cached on disk in byte form, and the block needs serializing before stored in the disk; and it needs de-serializing after loaded from disk vice versa. The corresponding storage scheme is `DISK_ONLY`. Obviously, the block does not take up memory space in this scenario, so it can work well in the case of memory insufficiency. The cost of revisiting a block consists of *Serialization Cost* and *Disk Cost*.

4.1.3 Memory and Disk Scenario

In this scenario, a block is cached either in memory or on disk. The corresponding storage scheme is `MEMORY_AND_DISK`. The place where a block is cached depends on free memory space at that time and block size. When the block could fit into memory, it would be cached in memory. Otherwise, it would be cached on disk. When the block is selected for eviction, it will be stored on disk as an alternative. On the contrary, when a block is being visited on disk, the decision on forwarding the block into memory depends on whether there is enough memory space to accommodate it.

4.1.4 Serialization Scenario

In this scenario, a block is serialized by configured serializer before being cached in memory. At this time, the value of δ in (4) is 1. And the value of β is less than 1 rather than 1 in memory scenarios. The corresponding storage scheme is MEMORY_ONLY_SER or MEMORY_AND_DISK_SER. As described in Sect. 1, it is an efficient way to reduce memory volume demands for data caching. As a consequence, the memory in this scenario may hold more blocks than it in memory scenarios with the same free memory space. Therefore it can greatly reduce the cost of re-computing and memory eviction. However, it comes with additional overheads for serializing.

4.2 Worst Case Execution Cost Model

As discussed above, storage memory space in Spark has two states: sufficient state and insufficient state. When a task finished in a sufficient memory state, we just consider *Visiting Costs* of all blocks in it. Otherwise, we should further consider *Creation Costs* of all blocks. Based on the above understanding, we present our worst-case execution cost model below:

$$\begin{aligned}
 C_{wcec}^k = & \sum_{b_{(i,j)} \in E} \left\{ (r_i^j - 1) * C_{(i,j)}^{vist} + C_{(i,j)}^{create} \right\} \\
 & + \sum_{b_{(i',j')} \in U} r_{i'}^{j'} * C_{(i',j')}^{create}
 \end{aligned} \tag{5}$$

where k is the stage id. E is a set of blocks that can be computed in a sufficient memory state in stage k . And U is a set of blocks that would be computed in an insufficient memory stage. Both E and U are from *Future Block Set* which we will introduce below. Here, r_i^j is the number of times that $b_{(i,j)}$ would be visited.

Future Block Set (FBS) For an executor l and a stage k , we define FBS as all un-computed *Cache Blocks* which would be computed in l at stage k , denoted as FB_l^k .

Initially, blocks in FBS can be achieved through *preferLocation()* API. For a node with multi executors, we equally divide the blocks to each executor. From (5), we find that blocks in FBS can be classified into two kinds: memory enough blocks and memory shortage blocks. For memory enough blocks, except for first, we just consider their *Visiting Costs* because these blocks would not be evicted before the stage finished. For memory insufficient blocks, they will be evicted before each revisiting. So these blocks need to be re-computed in each revisiting.

The worst-case execution cost of an application varies in different scenarios. *Visiting Cost* of a block is also different in each scenario. Similarly, memory space that a block demand is different. For example, a block occupies less memory space in serialization scenarios than in memory scenarios. So there will be more blocks that can be computed in a sufficient memory state in serialization scenarios. So they are in the disk scenario. As a consequence, we can conclude

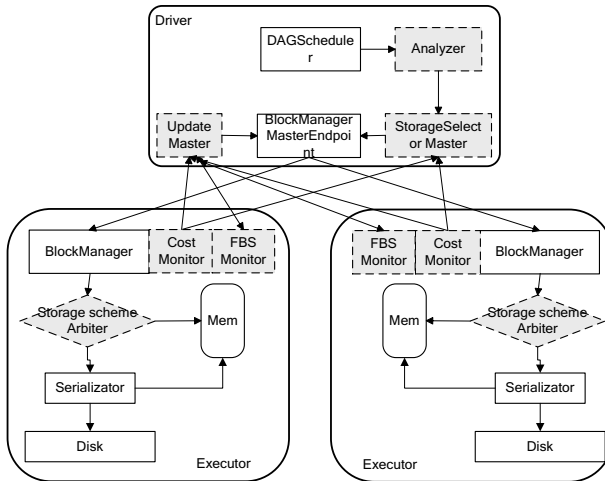


Fig. 5 Overall architecture of TFSS. Our components are marked with dash outlines

that the worst-case execution costs of the same executor are different in each scenario. So it is an opportunity for us to choose a proper storage scheme for blocks based on our predictions. The storage scheme, which has the least worst-case execution cost, will be chosen as the final scheme.

5 Implementation in Spark

We have extended Spark to support the TFSS mechanism presented in this article. And we will elaborate on our implementation details in this section.

5.1 Architecture Overview

TFSS can choose a storage scheme, based on worst-case execution cost and real-time memory state, for a block to be cached. As shown in Fig. 5, our implementation consists of four models: (i) *Analyzer*, which lies in the driver, learns the data dependency of each RDD and bookkeeps cache RDDs in each stage; (ii) *Cost Monitors*, which lie in each executor, are responsible for collecting *Cache Blocks*' costs information which is shown in Sect. 4 during the task running; (iii) *Storage-scheme Arbiters*, which also lie in each executor, implement logic policies that make storage scheme decision for each block before it will be cached; (iv) *Feedback components*, which consist of *Cost Monitors*, *FBS Monitors* and an *Update-Master*, are used for updating future executing blocks and costs information.

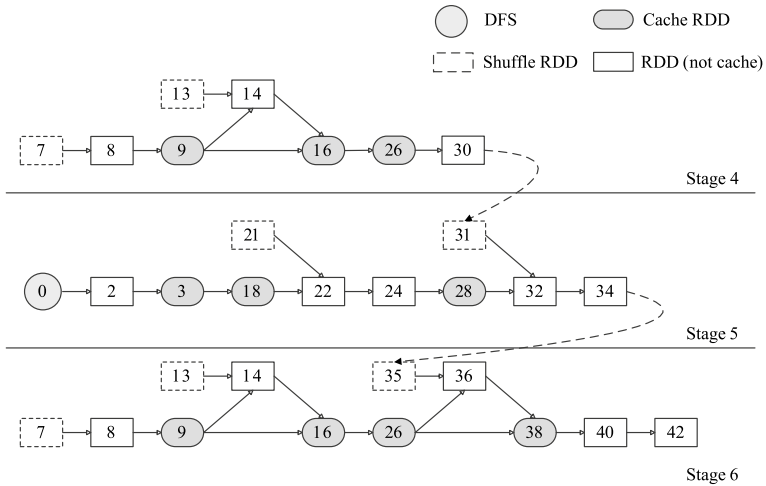


Fig. 6 Segment DAG of job 1 in PageRank

5.2 Analyzer

The *Analyzer* provides three functions: firstly, it obtains dependencies of RDDs according to DAG using a depth-first search; secondly, it also picks up cache RDDs in each stage; thirdly, it initializes FBS of each worker.

Figure 6 shows a segment of PageRank’s DAG. We can find that there are three stages. RDDs within each stage are connected through narrow dependencies. They stem from Distributed File System (DFS), Shuffled-RDDs, or other RDDs. Among them, files on DFS are on a local or remote node. The Shuffled RDDs are generated by tasks via fetching remote shuffle data. The workflow of a task may be the longest-running path when the task runs firstly. To avoid blocks being created with the longest path each time, Spark caches some RDDs. So there are two kinds of RDDs: cache RDDs, which can be directly revisited after the first computing, and non-cache RDDs which are only available in this calculation. Among them, cache RDDs are persisted in memory, disk, or both, according to their storage schemes which are assigned during programming, through *BlockManager*.

Cache RDDs can be further classified into three kinds: target CacheRDDs are the running cache RDDs in all stages; related CacheRDDs are RDDs that participate to create target CacheRDDs in the current stage, and other CacheRDDs are RDDs which are not including in the first two kinds. In Fig. 6, RDD 38 is a target CacheRDDs in stage 6, and RDDs 9, 16, and 26 are its related CacheRDDs. As well as RDDs 3, 18, and 28 are other CacheRDDs. In practice, we can use target CacheRDDs and its related CacheRDDs to infer *Computing Cost* and memory demands at a stage.

The Analyzer not only analyzes dependencies information between RDDs but also speculates future *Cache Blocks* that will be computed in each worker. After runnable tasks are submitted to the task scheduler, the task scheduler assigns each task

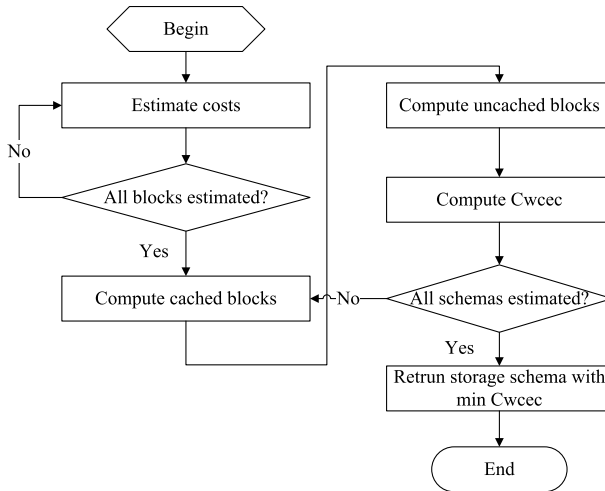


Fig. 7 Workflow of Storagescheme Arbiter

to the corresponding worker according to the scheduling policy. Usually, the task scheduler will put the tasks on the workers which have less burden, such as network I/O, to run. By default, Spark assigns tasks to workers using data locality policy. A worker has a high priority to get the task whose consuming blocks are on it. For a task, its preferred running workers will be returned by the scheduler through *prefer-Location()* API. Therefore, we can infer all future tasks running on a worker, and further on an executor. We suppose that tasks are allocated evenly to each executor if there are multi executors running on the worker. So the executor's FBS, which is used by *Storagescheme Arbiter*, can be got in this way.

5.3 Cost Monitors

Cost Monitors are distributed in each executor. A *Cost Monitor* collects local *Cache Blocks* real-time costs information. The information would be used by both *Storagescheme Arbiter* in this executor and *UpdataMaster* in the master. The *Cost Monitor* records the information of the costs when tasks perform related operations. As well as the information is passed to *UpdateMaster* with the help of the heartbeat. And the information of the costs would be recorded in a hash set, in which it is easy to be retrieved through a key of block Id.

5.4 Storage Scheme Arbiters

Storagescheme Arbiter gives *Cache Blocks* in this executor an appropriate storage scheme before they are cached. It makes a decision based on worst-case execution costs.

As shown in Fig. 7, *Storagescheme Arbiter* takes full consideration costs of *Cache Blocks* in FBS when it determines storage scheme for a *Cache Block*. As the discussion above, these blocks include those consumed by the tasks that are running and will run in the executor. The worst-case execution cost of the current stage in this executor is computed using (5). The arbiter selects a storage scheme that has minimum worst-case execution cost value when *BlockManager* caches a block. For example, it would choose a serialization scheme to improve efficiency if the worst-case execution cost under the serialization scenario is smaller than it under memory scenario. Otherwise, it would set the serialization flag to false. If the size of a block is larger than the size of free storage memory, it implies that the current memory is insufficient. If we cache the block in memory, it may fail to unroll due to *OOM*. At this time, a smart choice is putting the block on disk.

5.5 Feedback Components

As shown in Fig. 5, feedback components consist of *Cost Monitor*, *FBS Monitor*, and *UpdateMaster*. Among them, *Cost Monitor* has been introduced in the above, so we do not present it in this section.

5.5.1 UpdateMaster

FBS of each executor is initialized by *Analyzer* according to data locality, and is delivered by *UpdateMaster* when the executor is launched. Then there are three cases that we need to update FBS. Firstly, blocks in FBS would be consumed by itself or other executors during the job running. For example, executors on the same worker would consume some blocks for parallel execution; and executors on the other workers also consume some blocks for distributed multi-copy data storage. Secondly, the task scheduler may allocate other blocks, which are not in FBS, to the executor. Thirdly, some blocks need to be re-computed for executor crashing.

For consumed blocks, *UpdateMaster* maintains an *Allocated Blocks Set* (ABS) to record blocks having been allocated at current running stage. And the ABS is updated after a task allocation. Blocks, which are consumed by the new allocated task, are added in ABS. Then, master sends ABS to the executor through the heartbeat mechanism. Finally, FBS monitor in the executor does subtraction operation between FBS and ABS after receiving the message.

For non-preferred-location blocks that are not in FBS of an executor, *UpdateMaster* just delivers them to the executor when the task scheduler allocates the task. This updating process is relatively simple. The executor will do union operation between blocks that the new arrival tasks consume and FBS.

For re-computing blocks, *UpdateMaster* maintains a *Lost Blocks Set* (LBS) which records re-computing blocks consumed by lost tasks at current running stage. The corresponding blocks would be added into LBS after the executor crashed. Then, master sends LBS to the executor through the heartbeat mechanism. Finally, *FBS Monitor* in the executor does union operation between FBS and LBS after receiving the message.

Table 2 Summary of workload and input data size

| Workload | Input data size (GB) |
|---------------------|----------------------|
| KMeans | 11 |
| Linear regression | 11.5 |
| Logistic regression | 16.6 |
| Page rank | 1.7 |
| Connected component | 4 |
| Decision tree | 6 |
| Shortest paths | 4 |
| PCA | 18.4 |

Furthermore, *UpdateMaster* also takes charge of delivering costs information to *Cost Monitors*. It collects costs information from all executors and feedbacks them to *Cost Monitors* to update local costs information.

5.5.2 FBS Monitor

FBS Monitor in an executor maintains FBS in a HashMap. It updates FBS in two cases. On the one hand, it gets rid of the block from FBS after the executor finished a *Cache Block*; On the other hand, it updates local FBS in different operations depending on message kinds. As discussed above, it adds non-preferred-location blocks and re-computing blocks into FBS or removes consumed blocks from FBS when an updating message from *UpdateMaster* comes.

5.6 Overheads Discussion

Communication Overhead To reduce the communication overhead, each worker maintains a costs profile locally and sends the minimum number of message exchanges to the master. Generally, *UpdateMaster* updates the information to the corresponding worker, only when necessary, through the periodical heartbeat. In particular, when a new job DAG is received from *DAGScheduler*, *UpdateMaster* needs to notify workers to update the FBS and costs of the corresponding RDD blocks.

Computation Overhead There are three cases that the computation is needed: (1) when a new job DAG arrivals, *Analyzer* analyzes dependencies among RDDs in DAG; (2) when the framework begins to launch an executor, *Analyzer* gets FBS for the executor; (3) when *BlockManager* starts to cache a block, *Storagescheme Arbiter* should compute worst-case execution cost of different scenarios under real-time memory circumstance.

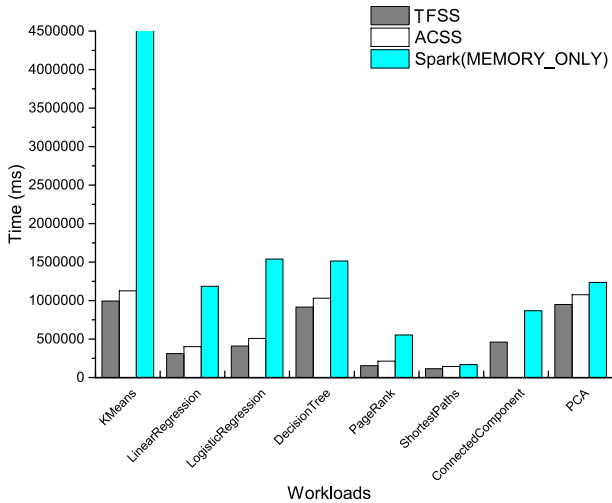


Fig. 8 Overall performance compares with different workloads

6 Performance Evaluations

6.1 Environment Platform

The experiment platform includes a cluster with four nodes, one acts as both master and slave and the remaining nodes only act as slaves. They are different from each other. The first type node is equipped with one 8-core 2.8G Intel Core I7 CPU, 12GB memory, the second type node is equipped with one 4-core 2.8G Intel Core I5 CPU, 8GB memory, the last type node is equipped with one 4-core 2.4G Intel Q6600 CPU, 8GB memory. And we adopt HDFS for storage, each partition has two replications. The JVM adopted is HotSpot 64-Bit Server VM, and the version is 1.8.0_92. The version of Spark is 2.3.0.

The datasets are generated by SparkBench [13]. Workloads, which we use in our experiments, are listed in Table 2.

6.2 Overall Performance

From the 1.6 version, Spark has supported the unified memory manager to manage storage memory and execution memory. So, we use the unified memory manager model in our experiments. And each executor uses 4 Cores and 2G memory size. Table 2 shows the input data size of each workload. We run each workload using our approach (denoted as TFSS), approach proposed in [9] (denoted as ACSS) and Spark native system with default storage scheme respectively. And the runtime of each workload is depicted in Fig. 8.

As expected, TFSS can reduce runtime by adjusting the cache storage scheme of *Cache Blocks* in time to prevent system performance decay. As shown in Fig. 8, our approach consistently outperforms the Spark MEMORY_ONLY scheme and ACSS

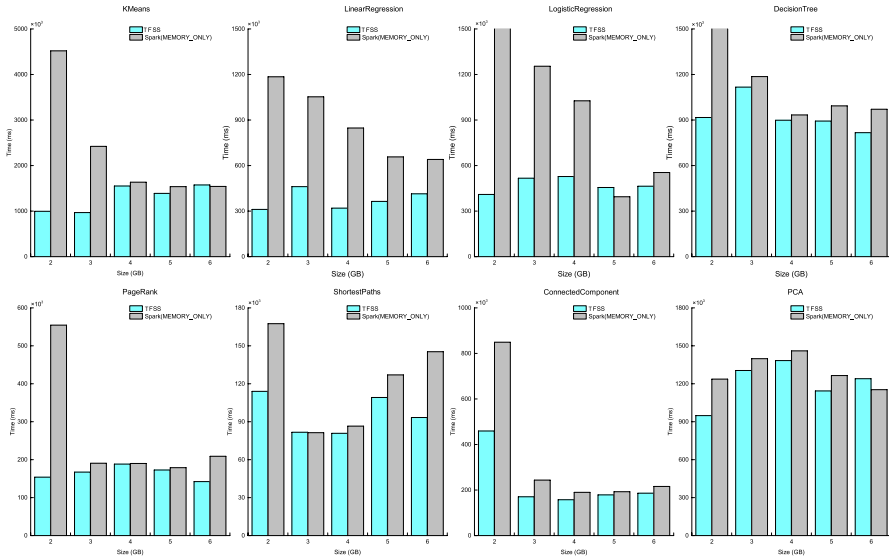


Fig. 9 Performance comparison between TFSS and MEMORY_ONLY storage scheme in different executor memory sizes

across all workloads. In particular, compared to the default storage scheme, TFSS can reduce the runtime of KMeans by 78% (from 4518 to 995 s). For ACSS, except for ConnectedComponent which fails to finish, TFSS can reduce the runtime by up to 27% (from 212 to 154 s) in PageRank.

6.3 Performance in Different Memory Sizes

In order to demonstrate the adaptability of our method under different memory constraints, we run eight workloads using the Spark default storage scheme and our storage scheme arbiter in different executor memory sizes. The memory size varies from 2GB to 6GB in each executor. The executor with a small memory size has a heavy constraint memory. As discussed above, heavy constrained memory brings performance decay for RDDs' re-computing, garbage collection, and so on. As shown in Fig. 9, the execution time of workloads increases with the decrease of memory size.

Additionally, Fig. 9 also shows the performance difference between TFSS and Spark native system under different constrained-memory degrees. The benefits achieved by TFSS, in terms of the application speedup, vary with different executor memory sizes as well as most workloads. For example, TFSS can reduce 35.4–73.7% execution time compared to MEMORY_ONLY scheme in different executor sizes in LinearRegression. Moreover, the benefits, achieved by TFSS, decrease with the increase of the execution memory size. Storage memory which can be used by cache RDDs increases with the increasing of executor memory size under a fix input data size. At the same time, the frequency of garbage collection as well as re-computation, caused by replacement, reduces with the increase of executor memory size.

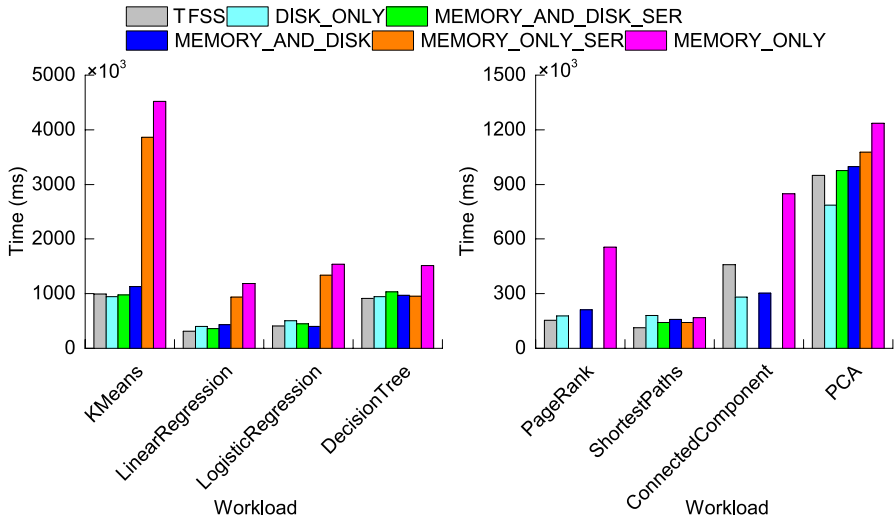


Fig. 10 Performance comparison between TFSS and Spark with different storage schemes in executor memory size 2 GB. Blank means failing

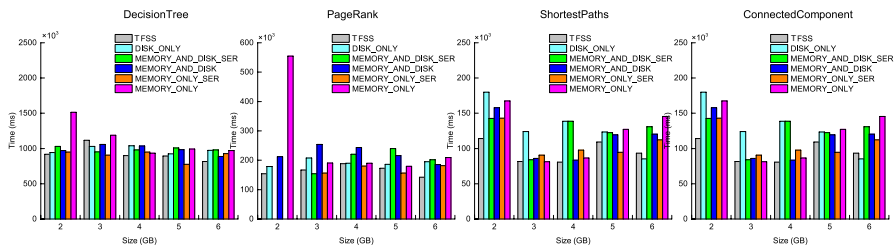


Fig. 11 Performance comparison between TFSS and Spark with different storage schemes with different executor sizes. Blank means failing

So the default scheme can work well under a slightly constrained-memory circumstance. TFSS also can catch this change. But performance improvement, made by TFSS, declines relative to a heavy constrained-memory circumstance.

6.4 Performance in Different Schemes

In order to demonstrate the adaptability of our method under different storage schemes, we run workloads using different storage schemes, which Spark provides, and in different executor memory sizes. The memory size also varies from 2 to 6 GB in each executor. We compare the workloads execution performance of TFSS with different storage schemes separately from two aspects.

First, we compare the performance of each workload under the different storage schemes in the most severe case of memory constraint. As shown in Fig. 10,

TFSS can work well in every workload. Instead, serialization schemes, MEMORY_ONLY_SER and MEMORY_AND_DISK_SER, fail in PageRank and ConnectedComponent. Furthermore, in addition to PCA and ConnectedComponent, our method achieves the optimal or near-optimal performance in all workloads. The results show that our method can correctly allocate suitable storage schemes for blocks. The fine-grained storage scheme allocation strategy can achieve better performance under certain circumstances, though the improvement is limited (almost 20% in ShortestPaths).

Second, we compare the performance of four workloads under the different storage schemes as well as different executor memory sizes. As shown in Fig. 11, TFSS can also work well in every workload with different executor memory sizes. The experimental results show that our method can choose the correct storage scheme not only under the serious constrained-memory circumstance but also under the sufficient memory circumstance. For example, TFSS achieves the optimal or near-optimal performance in PageRank across all executor memory sizes. And performance improves 21.58% against the best storage scheme, MEMORY_ONLY_SER, in size 6GB.

7 Related Works

Cache Management Memory caching is a popular approach with the aim to improve memory efficiency. Over the years, a large number of caching policies have been proposed.

The traditional history-based policies, such as LRU [15] and LFU [16], are widely used in prevalent computer systems. These algorithms, taking advantage of temporal locality and spatial locality, make a replacement decision based on data access history. Although it is simple to implement, it knows nothing about the data access information from applications. Hint-based policies use hints, which indicate which and when data will be used, from applications [17, 18]. Although they can improve cache efficiency with the help of hints, it is difficult for programmers to directly mark hints in the codes.

However, the above policies for managing cache data bring a significant performance loss in prevalent parallel computing frameworks [19, 20]. These systems exhibit different data access patterns compared with traditional systems. As described previously, workflows in these systems, mostly express in DAGs, are known before execution. So some policies take advantage of DAGs to infer data access patterns for guiding their decisions. For example, LRC [5] makes a decision according to blocks' reference counts, which come from DAG of the application. The blocks which have fewer reference counts are replaced from memory in priority. Further, more information is used for cache replacement. Except for reference times, information of creation time, serialization/deserialization time, I/O time and size are considered in their policies [21–24]. In these policies, DAGs are used to evaluate an RDD's future cost. RDDs with smaller cost value have precedence over being

evicted, and RDDs with greater cost value have precedence over being cached into memory.

Parameter Tuning There are many parameters in the parallel processing system, and the relationship between parameters and performance is nonlinear. So it's important to get the best parameter configuration by tuning. Existing approaches are divided into manual parameter tuning approaches [25] and automatic parameter tuning approaches two categories. Manual tuning approaches should be trial and error before getting a set of satisfying parameters, as well as users, have to go deep into comprehending the program and system that it runs on. Therefore it is time-consuming. Automatic tuning approaches have no such issues. Existing automatic tuning approaches are divided into three categories:

Approaches based on cost model. Cost-based approaches need users to create cost models based on the aim framework. For example, Wang et al. [26] creates a performance model with the job execution time of each stage based on the Spark workflow. Starfish [27, 28] employs a mixture of cost model and simulator to optimize a job based on previously executed job profile information. Compute spaces in these approaches are usually huge. So these approaches provide various ways to reduce compute time. For example, Starfish divides the configuration parameter space into subspaces. And MR-COF [29] uses a genetic algorithm to search the parameter space.

Approaches based on machine learning. These approaches train the performance model and search parameters space using machine learning algorithms. For example, Yigitbasi et al. [30] uses Support Vector Regression algorithm and Chen et al. [31] uses Tree Regression algorithm. Approaches in this category have good applicability and can be used based on different applications and clusters. However, there are significant impacts on system performance caused by the machine learning model.

Approaches based on searching. Search-based approaches search parameter space based on a fitness function using different search algorithms. For example, PRNN [32] provides parameters for recommended tasks by searching for similar tasks from a history database. Approaches of this kind mainly use a genetic algorithm to express the relationship among various parameters and improve search efficiency. For example, GEP [33] represents the relationship among configuration parameters with Gene Expression Programming and searches optimal or near-optimal parameters by the Particle Swarm Optimization algorithm. In addition, there are some online approaches that can automatically adjust task parameters. MRONLINE [34] uses a specific set of tuning rules based on the combination of the Intelligent Climbing algorithm to achieve online parameter adjustment. MRONLINE also supports fine-grained control of configuration parameters, using different configurations for each task. Ant [35] gets the optimal parameters after multiple waves of task execution. Parameters of the current tasks are generated through crossover and mutation on parameters of previous wave tasks. The online methods do not need to repeatedly perform and can dynamically adjust parameters according to the real-time tasks, so it is suitable for applications just running one or several times.

Storage Optimization MemTune [20] dynamically adjusts the heap share of execution memory and storage memory. However, it only considers dependencies

between currently running tasks. On the contrary, our approach takes full account of the entire DAG and dependencies. Koliopoulos [9] introduces a new view on improving the cache efficiency in Spark. It provides a storage scheme auto-selection mechanism basing on the relationship of data size and total memory size in a cluster. Different thresholds indicate corresponding storage scheme statically. Although it can successfully select a storage scheme for each RDD, the storage scheme is static in the whole running. In contrast, our approach can dynamically select a storage scheme for blocks.

8 Conclusions and Future Works

In this paper, we present a novel task-aware storage scheme selection mechanism named TFSS. When a job is running, TFSS calculates worst-case execution cost of different scenarios considering the current memory usage state and future computing task which would be executed in one executor of the current stage. Then, it automatically recommends an appropriate cache storage scheme with minimal cost. Compared with the default static specified storage scheme, TFSS can mitigate memory pressure and speed up the application execution. And differing from the RDD-Grained storage scheme selection mechanism used by Spark, our approach implements a task-grained storage scheme selection. Experiments show that when the memory space is limited the proposed approach can effectively improve the performance by up to 78%. Furthermore, our approach has better performance on the heterogeneous environment than the default mechanism. In the future, we plan to integrate a cost-based RDD cache replacement strategy to replace LRU which just uses history for replacement and does not take full advantage of known DAG execution flow.

Acknowledgements This work is supported by South China University of Technology Start-up Grant No. D61600470, Guangzhou Technology Grant No. 201707010148, The New Generation of Artificial Intelligence In Guangdong Grant No. 2018B0107003, Doctoral Startup Program of Guangdong Natural Science Foundation Grant No. 2018A030310408, The Key Science and Technology Program of Henan Province Grant No. 202102210152, and National Science Foundation of China under Grant Nos. 61370062, 61866038.

References


1. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. In: Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation, San Francisco, CA, p. 10 (2004)
2. Isard, M., Budiu, M., Yu, Y., et al.: Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, pp. 59–72. ACM, Lisbon, Portugal (2007)
3. Zhang, H., Chen, G., Ooi, B.C., et al.: In-memory big data management and processing: a survey. *IEEE Trans. Knowl. Data Eng.* **27**, 1920–1948 (2015). <https://doi.org/10.1109/TKDE.2015.2427795>

4. Zaharia, M., Chowdhury, M., Das, T., et al.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation. USENIX Association, San Jose, CA, p. 2 (2012)
5. Yu, Y., Wang, W., Zhang, J., et al.: LRC: dependency-aware cache management for data analytics clusters. In: Proceedings of INFOCOM, Atlanta, GA, USA, pp. 1–9 (2017)
6. Choi, I.S., Yang, W., Kee, Y.S.: Early experience with optimizing I/O performance using high-performance SSDs for in-memory cluster computing. In: Proceedings of IEEE International Conference on Big Data (Big Data 2015) (2015), pp. 1073–1083
7. Xin, R.: Project Tungsten (Spark 1.5 Phase 1). <https://issues.apache.org/jira/browse/SPARK-7075>. Accessed 1 Mar 2019
8. Geng, Y., Shi, X., Pei, C., et al.: LCS: an efficient data eviction strategy for spark. *Int. J. Parallel Program.* **45**, 1285–1297 (2017). <https://doi.org/10.1007/s10766-016-0470-1>
9. Koliopoulos, A.K., Yiapanis, P., Tekiner, F., et al.: Towards automatic memory tuning for in-memory Big Data analytics in clusters. In: Proceedings of IEEE International Congress on Big Data (BigData Congress 2016), pp. 353–356 (2016)
10. Saha, B., Shah, H., Seth, S., et al.: Apache Tez: a unifying framework for modeling and building data processing applications. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. ACM, Melbourne, Victoria, Australia, pp. 1357–1369 (2015)
11. Shvachko, K., Kuang, H., Radia, S., et al.: The Hadoop distributed file system. In: Proceedings of IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST 2010), pp. 1–10 (2010)
12. Zaharia, M., Borthakur, D., Sarma, J.S., et al.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European Conference on Computer Systems, pp. 265–278. ACM, Paris, France (2010)
13. Li, M., Tan, J., Wang, Y., et al.: SparkBench: a comprehensive benchmarking suite for in memory data analytic platform Spark. In: Proceedings of the 12th ACM International Conference on Computing Frontiers, pp. 1–8. ACM, Ischia, Italy (2015)
14. Zhao, Y., Hu, F., Chen, H.: An adaptive tuning strategy on spark based on in-memory computation characteristics. In: Proceedings of the 18th International Conference on Advanced Communication Technology (ICACT2016), pp. 484–488 (2016)
15. Mattson, R.L., Gecsei, J., Slutz, D.R., et al.: Evaluation techniques for storage hierarchies. *IBM Syst. J.* **9**, 78–117 (1970). <https://doi.org/10.1147/sj.92.0078>
16. Aho, A.V., Denning, P.J., Ullman, J.D.: Principles of optimal page replacement. *J. ACM* **18**, 80–93 (1971). <https://doi.org/10.1145/321623.321632>
17. Cao, P., Felten, E.W., Karlin, A.R., et al.: Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. *ACM Trans. Comput. Syst.* **14**, 311–343 (1996). <https://doi.org/10.1145/235543.235544>
18. Patterson, R.H., Gibson, G.A., Ginting, E., et al.: Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.* **29**, 79–95 (1995). <https://doi.org/10.1145/224057.224064>
19. Ananthanarayanan, G., Ghodsi, A., Wang, A., et al.: PACMan: coordinated memory caching for parallel jobs. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, p. 20. USENIX Association, San Jose, CA (2012)
20. Xu, L., Li, M., Zhang, L., et al.: MEMTUNE: dynamic memory management for in-memory data analytic platforms. In: Proceedings of IEEE International Parallel and Distributed Processing Symposium, pp. 383–392 (2016)
21. Feng, L.: Research and implementation of memory optimization based on parallel computing engine spark. M.S. Dissertation, Tsinghua University, China (2013)
22. Wang, K., Zhang, K., Gao, C.: A new scheme for cache optimization based on cluster computing framework spark. In: Proceedings of 8th International Symposium on Computational Intelligence and Design (ISCID 2015), pp. 114–117 (2015)
23. Duan, M., Li, K., Tang, Z., et al.: Selection and replacement algorithms for memory performance improvement in Spark. *Concurr. Comput. Pract. Exp.* **28**, 2473–2486 (2016)
24. Bian, C., Yu, C., Ying, C., et al.: Self-adaptive strategy for cache management in spark. *Chin. J. Electron.* **45**, 278–284 (2016)
25. Spark Tuning. Online Referencing. <http://spark.apache.org/docs/latest/tuning.html/#tuning-spark>. Accessed 1 Mar 2019
26. Wang, G.L., Xu, J.G., Liu, R.F.: A performance automatic optimization method for spark. Patent 105868019 A, CN (2016)

27. Herodotou, H., Babu, S.: Profiling, what-if analysis, and cost-based optimization of MapReduce programs. *VLDB* **4**, 1111–1122 (2011)
28. Herodotou, H., Lim, H., Luo, G., et al.: Starfish: a self-tuning system for Big Data analytics. In: *Proceedings of Fifth Biennial Conference on Innovative Data Systems Research (CIDR2011)*, Asilomar, CA, USA, pp. 261–272 (2011)
29. Liu, C., Zeng, D., Yao, H., et al.: MR-COF: a genetic MapReduce configuration optimization framework. In: *Proceedings of the 15th International Conference of Algorithms and Architectures for Parallel Processing (ICA3PP 2015)*, pp. 344–357(2015)
30. Yigitbasi, N., Willke, T.L., Liao, G., et al.: Towards machine learning-based auto-tuning of MapReduce. In: *Proceedings of the 2013 IEEE 21st International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems*, pp. 11–20 (2013)
31. Chen, C.O., Zhuo, Y.Q., Yeh, C.C., et al.: Machine learning-based configuration parameter tuning on Hadoop system. In: *Proceedings of IEEE International Congress on Big Data*, pp. 386–392 (2015)
32. Chen, Q.A., Feng, L., Yue, C., et al.: Parameter optimization for spark jobs based on runtime data analysis. *China Comput. Eng. Sci.* **38**, 11–19 (2016)
33. Khan, M., Huang, Z., Li, M., et al.: Optimizing Hadoop parameter settings with gene expression programming guided PSO. *Concurr. Comput. Pract. Exp.* (2017). <https://doi.org/10.1002/cpe.3786>
34. Li, M., Zeng, L., Meng, S., et al.: MRONLINE: MapReduce online performance tuning. In: *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, pp. 165–176. ACM, Vancouver, BC, Canada (2014)
35. Cheng, D., Rao, J., Guo, Y., et al.: Improving MapReduce performance in heterogeneous environments with adaptive task tuning. In: *Proceedings of the 15th International Middleware Conference*, pp. 97–108. ACM, Bordeaux, France (2014)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Affiliations

Bo Wang¹ · Jie Tang²  · Rui Zhang³ · Jialei Liu¹ · Shaoshan Liu⁴ · Deyu Qi²

Bo Wang
bowang@aynu.edu.cn

Rui Zhang
zr@yau.edu.cn

Jialei Liu
01850@aynu.edu.cn

Shaoshan Liu
shaoshan.liu@perceptin.io

Deyu Qi
csqideyu@scut.edu.cn

¹ Anyang Normal University, Anyang, China

² South China University of Technology, Guangzhou, China

³ Yan'an University, Yan'an, China

⁴ PerceptIn, Shenzhen, China