Computers, Materials & Continua
DOI:10.32604/cmc.2021.016462
Article

Tech Science Press

Improving Cache Management with Redundant RDDs Eviction in Spark

Yao Zhao[1], Jian Dong[1,*], Hongwei Liu[1], Jin Wu[2] and Yanxin Liu[1]

[1]School of Computer Science and Technology, Harbin Institute of Technology, Harbin, 150001, China
[2]School of Engineering, University of Georgia, Athens, 30602, USA
[*]Corresponding Author: Jian Dong. Email: dan@hit.edu.cn
Received: 02 January 2021; Accepted: 03 February 2021

Abstract: Efficient cache management plays a vital role in in-memory data-parallel systems, such as Spark, Tez, Storm and HANA. Recent research, notably research on the Least Reference Count (LRC) and Most Reference Distance (MRD) policies, has shown that dependency-aware caching management practices that consider the application's directed acyclic graph (DAG) perform well in Spark. However, these practices ignore the further relationship between RDDs and cached some redundant RDDs with the same child RDDs, which degrades the memory performance. Hence, in memory-constrained situations, systems may encounter a performance bottleneck due to frequent data block replacement. In addition, the prefetch mechanisms in some cache management policies, such as MRD, are hard to trigger. In this paper, we propose a new cache management method called RDE (Redundant Data Eviction) that can fully utilize applications' DAG information to optimize the management result. By considering both RDDs' dependencies and the reference sequence, we effectively evict RDDs with redundant features and perfect the memory for incoming data blocks. Experiments show that RDE improves performance by an average of 55% compared to LRU and by up to 48% and 20% compared to LRC and MRD, respectively. RDE also shows less sensitivity to memory bottlenecks, which means better availability in memory-constrained environments.

Keywords: Dependency-aware; cache management; in-memory computing; spark

1 Introduction

With the increasing demand for data analytics, in-memory data-parallel systems, such as Spark [1], Tez [2], HANA, and Storm [3], have shown advantages in iterative data processing with lower latency [4–7]. These in-memory frameworks lead to great performance improvements compared with disk-based frameworks and have become popular in industry. However, even with the lower prices of RAM, memory remains a constrained resource as the amount of data grows in

big data analytics [6,8]. Therefore, cache management has a crucial influence on the performance of in-memory data-parallel systems.

Cache optimization has been widely studied, and many efficient algorithms have been proposed to satisfy various systems [9]. Traditional cache management focuses on improving the hit ratio using certain prediction methods. However, traditional cache management is oblivious to data dependencies and shows poor performance in in-memory data-parallel systems [5]. In typical data-parallel systems, data dependency can be concluded before the execution of jobs by analyzing the structure of directed acyclic graphs (DAGs) [10]. Tasks in these parallel systems are executed in a determined workflow according to the DAGs, which can be exploited for scheduling and data caching [11,12].

Recent studies show that a cache policy considering data dependencies in data-parallel systems has a better performance than traditional history-based methods. Several dependency-aware cache policies for Spark, such as LRC [13], LCRC [14] and MRD [15], have been proposed. All of these policies have led to progress in improving the cache hit ratio compared to the default LRU cache policy in Spark [16]. LRC traverses the DAG and sets each Resilient Distributed Dataset (RDD) with different caching priorities according to its reference count, and RDDs with low reference counts tend to be evicted when memory is full. LCRC and MRD further exploit the DAGs of jobs and consider the reference gap, which makes the cached RDD more time-sensitive and achieves a better hit ratio than LRC. However, all these cache policies neglect the fact that RDDs with certain dependencies always share similar priorities in these algorithms. These RDDs tend to be cached together but play the same role in computing, which results in performance degradation when memory is a constrained resource.

In this paper, we discuss how the DAG can be further exploited to optimize cache management. The solution should traverse the DAGs of applications and implement DAG-based cache management with an efficient redundant block eviction strategy. Moreover, the policy should have low overhead and be applicable to DAG-based in-memory data-parallel computing systems.

We propose a novel cache management policy, Redundant Data Eviction (RDE), that can release more available memory space with low overhead. RDE can find the deeper relationships between data blocks and evict redundant blocks as a function. Furthermore, with the memory space freed by evicting redundant data, we launch a prefetching mechanism in cache management for further performance improvement. RDE has the following advantages:

First, RDE can minimize the caching blocks by evicting target redundant data blocks. We analyze mass DAGs of typical applications to exploit the features of redundant data blocks. As a result, we can precisely target redundant data using RDDs' dependencies and the schedule sequence in the workflow of applications. By evicting these redundant data, systems will have more memory space for computing and data caching, which will surely improve the performance.

Second, a cache management policy with redundant data eviction is more likely to attach a prefetching policy to achieve a better hit ratio in future workflows. As mentioned above, memory is always a constrained resource in data-parallel systems. RDE has less memory sensitivity and could have a better performance in resource-strict situations compared to previous cache management policies.

We implement RDE as a pluggable memory manager in Spark 2.4. To verify the efficiency of RDE, we conduct extensive evaluations on a six-node cluster with ten different data analysis workloads. For all the benchmarks, RDE shows high performance and large advantages in memory-constrained situations. According to our experimental results, RDE reduces the

application runtime by 41% on average compared with the default LRU caching policy in Spark and generally improves the performance of the system by 35% and 20% compared to LRC and MRD, respectively.

The structure of the remainder of this paper is organized as follows. Section 2 presents the background and describes the inefficiency of existing cache strategies based on DAGs derived from system schedulers. The design of RDE and its implementation details are proposed in Section 3. The evaluation results are reported in Section 4. Finally, we conclude the paper in Section 5.

2 Background and Motivation

In this section, we discuss the background of data access in Spark jobs and provide the motivation for introducing a novel cache management policy. We limit our discussion to the context of Spark in this paper. However, the discussion is also applicable in other in-memory computing frameworks.

2.1 RDD and Data Dependency

Spark is a distributed, in-memory computing framework for big data that provides the Resilient Distributed Dataset (RDD) as its primary abstraction in computing. RDDs are distributed datasets stored in memory. Spark can only transform an RDD into a new RDD using a transformation operation. The workflows of data on parallel computing frameworks are determined by DAGs consisting of RDDs. These DAGs contain rich information on data dependencies, which is crucial for data caching and has not been fully explored in default cache management policies.

For example, as a key abstraction in Spark, an RDD is a collection of objects partitioned across nodes in a Spark cluster [17], and all the partitions can be computed in parallel. In Spark, operations are divided into transformations and actions, and all the operations are based on RDDs. As shown in Fig. 1, the scheduler of Spark is composed of RDD objects, a DAG scheduler, task scheduling and task operations. During the construction of the RDD objects, the scheduler will analyze the RDDs of upcoming tasks and submit them to the DAG scheduler while the action operation is triggered. Then, the DAG scheduler forms a DAG by implying a task execution sequence that is divided into several stages. During the execution, the dragging or failing tasks are recomputed. Therefore, cache replacement strategies have significant influences on recomputing costs.

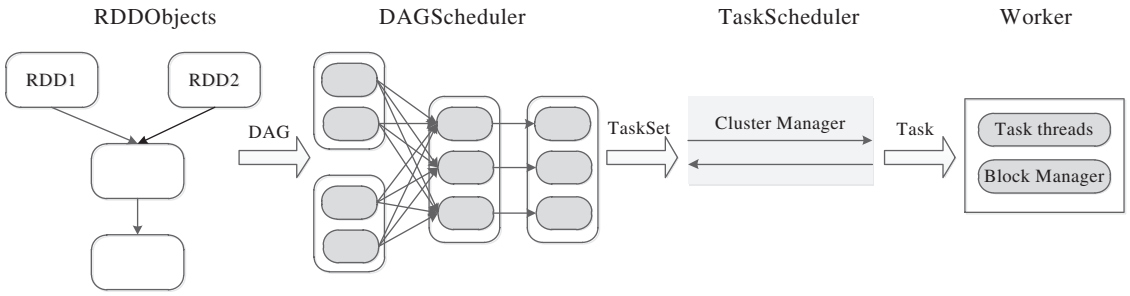

Figure 1: Schedule in spark

2.2 Memory Management in Spark

In Spark, memory is divided into three parts: System memory (other and system reserved), execution memory and storage memory, as shown in Fig. 2 [18]. RDDs are cached in storage memory. A uniform memory management mechanism was implemented in Spark after version 1.6 was updated. The execution memory and storage memory share the same memory pool, and their space can be dynamically changed to satisfy different memory requirements. This mechanism reduces the difficulty of managing the memory. This means that the utilization of storage memory will also have an influence on the available execution memory, which is responsible for the computing efficiency of applications [19,20].

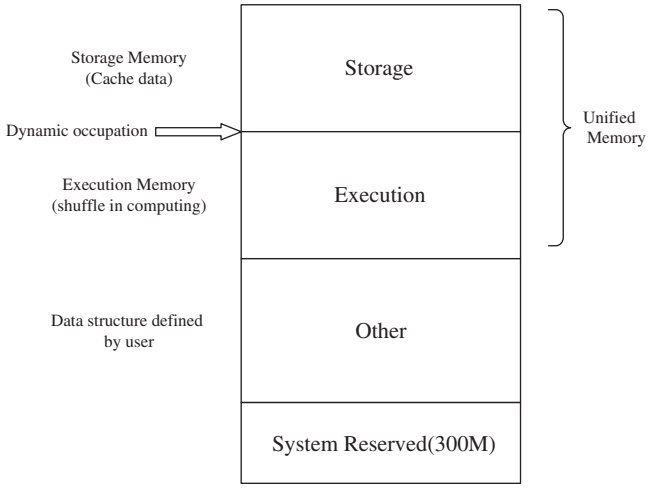

Figure 2: Memory management model of spark

We apply SparkBench [21] with different workloads on our cluster and analyze the logs of the system to explore the relationship between the storage memory and execution memory. The system logs show that when the storage memory is full and the execution memory drops to the minimum percentage, the system experiences degraded performance because of the frequent block replacement in computing RDDs. In some memory-constrained situations with heavy workloads, the RDD in computing even evicts its own blocks from the execution memory due to their low priority in existing DAG-based cache polices.

2.3 History-Based and DAG-Based Cache Management

History-based cache management is widely used in various systems. LRU is a classic history-based cache management method and is used as the cache replacement algorithm in Spark. LRU keeps tracking the data in memory and evicts the blocks that have not been accessed for the longest periods of time. However, LRU is oblivious to the lineage of Spark jobs, resulting in poor efficiency of the eviction of RDDs.

To fully utilize the DAGs and achieve a more significant performance improvement, several DAG-based cache management policies have been proposed. LRC and MRD are both representative DAG-based cache policies and have been proven to have high performance on common benchmarks. LRC traverses the lineage and keeps tracking the dependency count of each RDD. This count is updated continuously as a priority for evicting RDD blocks from memory as the

Spark jobs run. An RDD with a higher reference count is more likely to be used in future computations and should be cached in memory. To save the maximum amount of memory space, the RDD with the lowest dependency count should be evicted from memory. Compared with the default LRU policy, LRC improves the cache hit ratio and presents a better comprehensive application workflow. MRD analyzes the shortness of LRC and aims at improving the time sensitivity for caching. MRD always evicts data blocks whose reference distances are the largest and prefetches the data blocks with the lowest reference distances if possible. MRD performs better than LRC in systems with efficient memory.

However, when it is difficult to conduct prefetching and memory is constrained, frequent data block replacement will result in a significant performance degradation in MRD, while LRC can still obtain a better performance improvement on the system. Existing cache policies neglect the waste of memory resulting from redundant RDDs.

For example, in the lineage of the Connected Component (CC) shown in Fig. 3, RDD12 and RDD16 always appear in the same stage, and RDD16 is a child of RDD12 according to the dependency. This means that in most situations, RDD12 is redundant when RDD16 has been cached. However, in existing cache policies, including LRC and MRD, RDDs with the features mentioned above generally share similar cache priorities. These RDDs usually have high priority and are hard to evict throughout the workflow, which will reduce the space for the storage memory and prevent the allocation of more space to the execution memory because of the dynamic memory management in Spark. With an efficient redundant RDD eviction strategy, more memory space will be released for data caching and RDD prefetching. We further examine the CC's lineage. According to the reference distance first policy defined by the MRD policy, RDD9, RDD12 and RDD16 have the same high caching priority in the workflow and are hard to replace with other RDDs. RDD3, RDD14 and RDD22, which are also crucial RDDs in the workflow, are hard to cache due to the constrained memory.

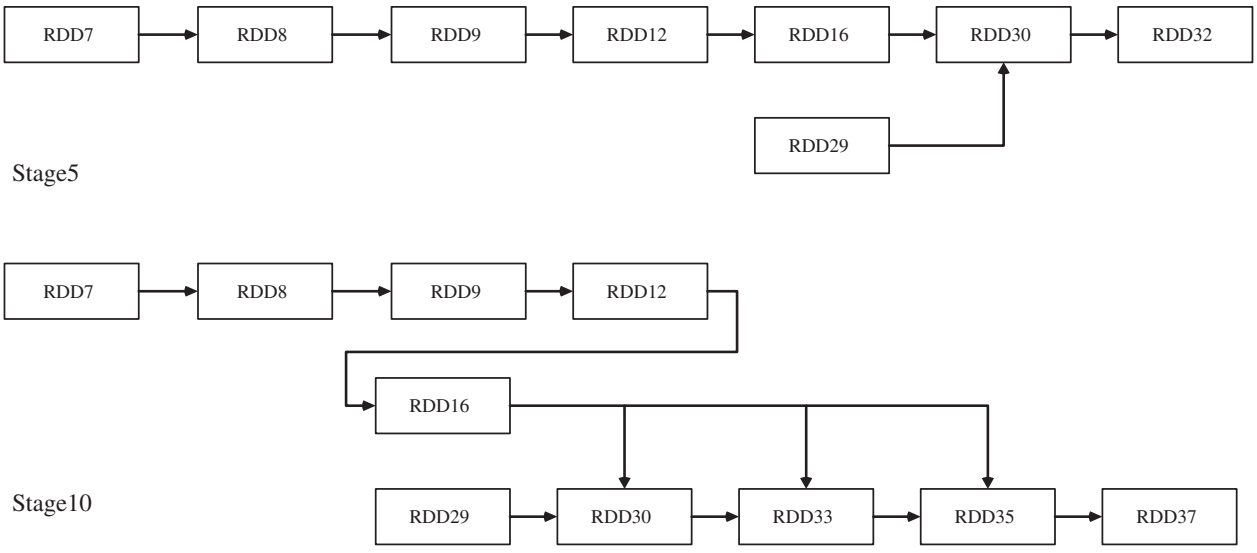

Figure 3: Partial lineage of a connected component (CC)

3 System Design

In this section, we propose a new cache management method, RDE (Redundant Data Eviction), which can make cache decisions based on the DAGs of applications with an efficient redundant RDD eviction policy. We also describe our implementation in Spark.

3.1 Eviction Policy in RDE

Definition 1 (Candidate RDDs): RDDs in a stage with the same computing sequence are defined as candidate RDDs.

The computing sequence of RDD_i represents the stage sequence including RDD_i in the workflow, and it is denoted as $CS(RDD_i)$. $CS(RDD_i)$ is described as the distance between the present stage and the next stage, including RDD_i. If the present stage is the last stage including RDD_i, then $CS(RDD_i)$ is considered to reach infinity.

Definition 2 (Redundant RDD): An RDD is called a redundant RDD in a candidate RDD set if and only if the RDD is not the leaf RDD after implementing a depth-first search in the candidate RDD workflow, which is derived from the DAG of the application.

The two definitions compose the criterion for locating redundant RDDs in a DAG derived from the Spark scheduler. RDE is a DAG-based cache management policy with an efficient redundant RDD eviction strategy. Each RDD in a DAG has two parameters: The basic cache priority and the CS. We first traverse the DAG and compute each RDD's CS according to the stage distance, which is utilized to represent the RDD computing sequence. In each stage, we perform a depth-first search among candidate RDDs with the same CS, only preserve the leaf RDD as the cache candidate, and view the remaining RDDs as redundant RDDs according to Definition 2. Then, we set the redundant RDDs with the lowest cache priority and recreate a new DAG to provide other RDDs with new cache priority. To coordinate with the prefetching algorithm, we use the stage distance to measure the basic cache priority of each RDD. For example, in Stage 6 of the PageRank lineage (Fig. 4), RDD9, RDD14, RDD16, RDD26, RDD35, RDD36 and RDD38 have the same CS, which is 9, and can be seen as candidate RDDs in Stage 6. After performing a depth-first search among candidate RDDs according to the dependency derived from the DAG, RDD16 is reserved as a candidate caching RDD. The others will be seen as redundant RDDs, and their cache priority will be set as the lowest. Stage 3, Stage 4 and Stage 15 share similar processes in the lineage of PageRank. This cache policy is oblivious to redundant RDDs, which is surely beneficial for future caching and prefetching. RDE shows advantages in memory utilization.

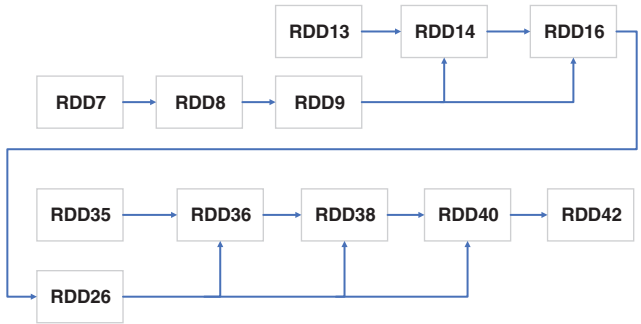

Figure 4: Lineage of stage 6 in pagerank

RDE can locate redundant RDDs quickly and avoid unnecessary overhead in memory. RDE provides systems with higher performance in memory-constrained situations. Moreover, the systems log of various workloads shows that the memory occupation remains at a high level due to Spark's efficient memory management in all stages, which means that the prefetching process in existing cache management policies may lead to frequent block replacement and degrade the performance in Spark. RDE is less memory sensitive and can cache more essential RDDs. RDE will decrease the frequency of prefetching and reduce the overhead introduced by the prefetching mechanism in Spark. The eviction method can be described using the following algorithm.

Algorithm 1: Eviction in RDE

```
Input: DAG of application
RDD_CS_table:a table with CS of RDDs in DAG
RDD_disk_CS:a table with CS of RDDs in disk
//locate redundant RDDs
for each RDD_i of DAG do
        RDD_i.CS ← stage distance from DAG
      end for
       for each stage m of DAG do
          sort RDDs based on RDD.CS
             CandidateRDDs m ← RDD with same CS
          do depth-first search in CandidateRDDs m
           if RDD_i in CandiateRDDs m is not leaf RDD
                 RedundantRDDs ← RDD_i
              end if
          end for
           for each RDD_i in DAG removed Redundant RDDs
          update(RDD_CS_table) //update CS of RDDs
          end for
//Block Eviction
          if data block size of (RDD_i) > free memory do
             RDD_j ← highest(RDD_CS_table)
             if(RDD_j · CS > RDD_i · CS)
                If(RDD_j · CS is not infinity)
                  evict(RDD_j)
                  write RDD_j to Disk
                 update(RDD_disk_CS)
            else
                  evict(RDD_j)
                  end if
             end if
          end if
```

3.2 Prefetching Mechanism

As mentioned above, we choose the computing sequence to measure the cache priority for each RDD. Computation in Spark occurs when a new stage is established. The RDD computing sequence can be represented by the stage computing sequence. Each RDD has various stage

distances that represent the different schedule orders in the entire workflow. An RDD with a noninfinite CS implies that this RDD will be used in future computations in current applications, and we consider this RDD to be a prefetchable RDD. However, prefetchable RDDs with lower cache priority need to be written to a disk to make room for higher priority RDDs. We keep a computing sequence table for prefetchable RDDs written to a disk. When the cache priority of prefetchable RDDs increases as a job runs and becomes higher than that of cached RDDs, the prefetching mechanism begins to work and cache prefetchable RDDs from the disk. RDE shows less memory sensitivity in computing, and the frequency of prefetching is lower than that of existing cache policies. RDDs with high priorities have less opportunity to be evicted by a prefetching mechanism, which will surely reduce the overhead caused by introducing a prefetch mechanism. The RDE cache management policy with prefetching can be described by the following algorithm.

Algorithm 2: RDE with prefetching

```
1: Input: free_memory
2: //prefetch RDD
3:        for each stage i in computing
4:          do Eviction in RDE
5:          update(RDD_CS_table)
6:          update(free_memory)
7:          RDD_i ← lowest(RDD_disk_CS)
8:          RDD_j ← highest(RDD_CS_table)in Cache
9:         for(RDD_i · size > free_memory)
10:          if(RDD_i · CS < RDD_j · CS)
11:            evict(RDD_j)
12:            RDD_j ← highest(RDD_CS_table)in Cache
13:            update(free_memory)
14:          end if
15:         end for
16:          prefetch(RDD_i)
17:       end for
```

We observe that to achieve basic cache priority, we need to traverse the entire application's DAG. However, in systems such as Spark, applications usually consist of several jobs, and we can only obtain the DAG of the present job from the Spark scheduler. Therefore, it is a challenge to achieve the entire DAG of applications. To solve this problem, we reconsider our cache policy in two situations.

Mostly, applications that run on in-memory data-parallel systems are recurring and usually repeat certain jobs with the same DAG to process different data sets. Therefore, it is feasible to learn the entire DAG from previous jobs so that our cache policy performs better in these applications.

For nonrecurring applications with jobs that have different DAGs, in each single job, RDE works in the same way as in recurring applications, but the redundant RDDs and the cache priority should be recomputed when a new job is coming. The hit ratio will drop by a certain percentage compared with recurring applications.

3.3 Spark Implementation

Architectural overview. Fig. 5 shows the architectural overview and the interaction between the modules of the cache manager. Our implementation is composed of 3 pluggable modules: DAGAnalyzer and REManager are deployed on the master node and CacheMonitor is deployed on each slave node. The other modules, such as DAGScheduler, BlockManager EndpointMaster and BlockManager SlaveEndpoint, are original components of Spark. The details of the main APIs for our implementation are given in Tab. 1.

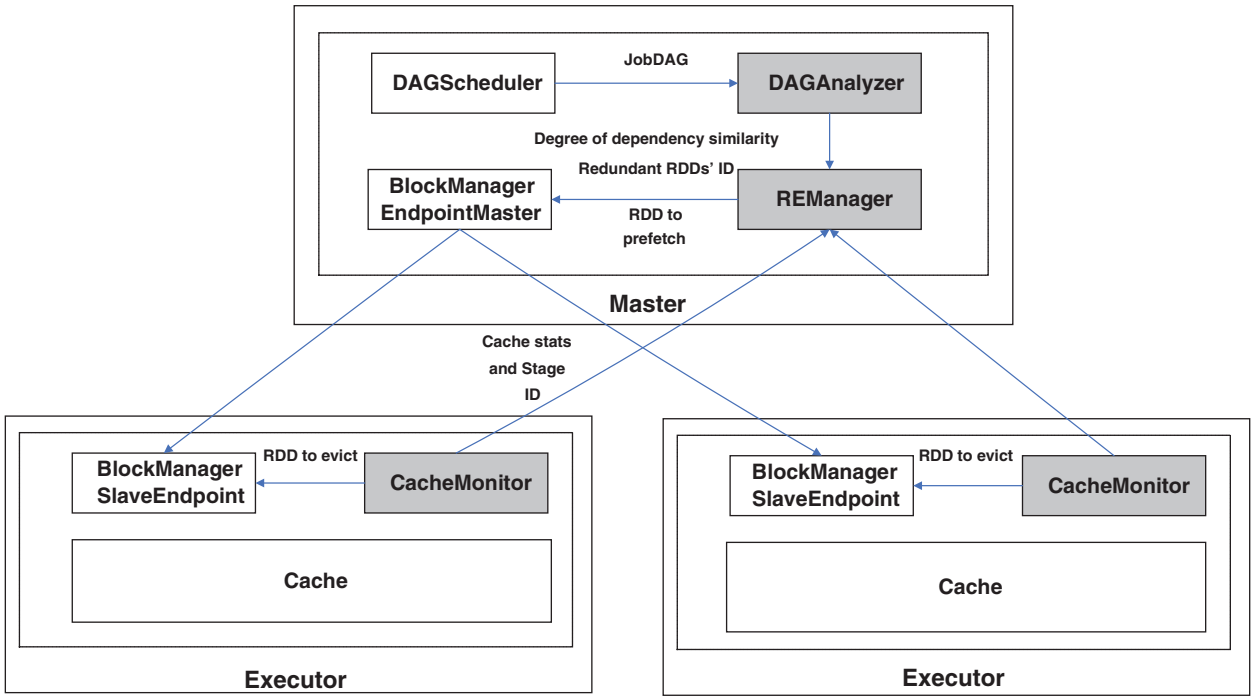

Figure 5: Overall system architecture of spark with RDE cache management. Our modules are highlighted as shaded boxes

DAGAnalyzer. DAGAnalyzer derives a job's DAG from the Spark DAGScheduler to prepare the essential information for REManager. In recurring applications, DAGAnalyzer creates the entire DAG of the application by analyzing the previous job's DAG. Then, it analyzes the DAG and calculates the CS for each RDD. Finally, DAGAnalyzer sends the application's DAG together with the RDDs' computing sequence to REManager.

REManager. REManager is the key component of this architecture. REManager reconstructs the application's DAG according to the information received from DAGAnalyzer. After updating the DAG by re-evaluating redundant RDDs, REManager recomputes the cache priority for other RDDs according to their stage distance in the new DAG. Moreover, with the information collected by CacheMonitor deployed on the slave nodes, REManager is also responsible for RDDs' eviction and prefetching algorithm at runtime.

CacheMonitor. CacheMonitors are deployed on the slave nodes in the cluster. CacheMonitors access various APIs and collect necessary information for data eviction and prefetching. Moreover,

CacheMonitors also conduct the RDD eviction strategy according to the instructions sent back from REManager.

Table 1: APIs of spark implementation

API	Description
DAGProfile	REManager reconstructs the DAG and return cache priority for each RDD by recomputing the stage distance
updateCachePriority	REManager sends a new cache priority file to CacheMonitor
updateDSD	DAGAnalyzer returns a new dependency similarity index when receiving new DAGs
BlocksEviction	When the cache is full, data with low cache priority will be evicted
DataPrefetch	Prefetch specific blocks used in the next stage

Workflow. After submitting the application, the Spark driver creates a SparkContext within which DAGAnalyzer and REManager are established. Meanwhile, other modules in Spark are also established. The driver then informs the slave nodes to launch the Spark executors and then deploys CacheManager and BlockManager. After establishing the connection between the driver and the executor, DAGAnalyzer analyzes the job's DAG from DAGScheduler, and the entire application's DAG together with the CS of each RDD are sent to REManager. REManager reconstructs the DAG and recomputes the RDDs' cache priority. Combined with the running information received from CacheMonitor and the new cache priority, REManager sends the eviction and prefetching strategy of present stages to BlockManager MasterEndpoint. Then, BlockManager communicates with BlockManager MasterEndpoint to conduct the specific cache operation, such as evicting RDDs from the cache or conducting prefetching by pulling data from the disk.

Communication overhead. RDE results in a slight communication overhead for Spark. While REManager reconstructs the DAG and determines the cache priority for each RDD, the cache priority file is sent to each slave node and can be kept locally during the workflow. REManager only updates the cache priority when necessary through heartbeats between the master and slave nodes. Specifically, REManager should inform the slave nodes to update their initial RDD caching priority when DAGAnalyzer receives a new job's DAG. Thus, the overhead from communication could be neglected during this workflow.

Prefetching overhead. In most situations, prefetching will surely improve the system's performance by increasing the cache hit ratio. Only in some extreme memory-constrained systems does prefetching cause frequent data replacement, and data with high priority, which should be computed in the next stage, could be evicted from the cache due to severe memory occupation. RDE evicts redundant RDDs and is more unlikely to experience a memory bottleneck compared to existing DAG-based cache polices. Therefore, the prefetching overhead could be ignored.

4 Evaluations

In this section, we evaluate the performance of our cache policy with typical benchmarks.

4.1 Experimental Environment

Our experimental platform was composed of several virtualized machines in two high-performance blade servers, which had 32 cores and 64 GB of memory each. The main tests were conducted in this virtual environment with nine nodes, which consisted of one master and eight slave nodes. The master node obtained a better configuration to satisfy the computing demand for cache policies. All the nodes were deployed with Spark 2.4.0 and Hadoop 2.8.0. The datasets were generated by SparkBench. The workloads and the amounts of input data are given in Tab. 2.

Table 2: Workloads and data input

Workloads	Amounts of data (GB)
KMeans	10
PageRank	11
Connected component	8
PregelOperation	7
SVD++	8.3

4.2 Overall Performance

The master is configured with 8 cores and 8 GB of memory, while the slave nodes are configured with 4 cores and 4 GB of memory. We compared RDE with the Spark native cache policy LRU and two typical DAG-based cache policies, known as LRC and MRD. We show the results for two different scenarios: RDE with eviction-only and RDE with both eviction and prefetching. In both scenarios, RDE performs well, especially with prefetching; and RDE significantly decreases the benchmark runtime by increasing the hit ratio of the cache.

RDE with eviction-only. We conducted RDE eviction-only on several Spark benchmarks and compared its performance with the performances of the LRU, LRC and MRD polices. The results are shown in Fig. 6.

It is clear that the application runtimes are reduced by up to 56% compared to the original cache policy LRU. Furthermore, RDE considers both the dependency and computing sequence of RDD, is more time sensitive than LRC in caching RDDs and achieves as high as a 30% improvement in performance over the Connected Component (CC) workload. RDE with the eviction-only policy also has a 9% to 15% performance improvement compared to MRD with the eviction-only policy since RDE caches less redundant RDDs to obtain a better hit ratio. In general, RDE provides a significant performance improvement due to its efficient eviction policy.

RDE with prefetching. RDE evicts redundant RDDs to free memory space for more valuable RDDs. With this mechanism, RDE can be more suitable for prefetching policies in memory-constrained situations. We conducted RDE with prefetching on the same benchmarks and datasets above, and the results of this method compared with those of LRU, LRC and MRD-Prefetch are shown in Fig. 7.

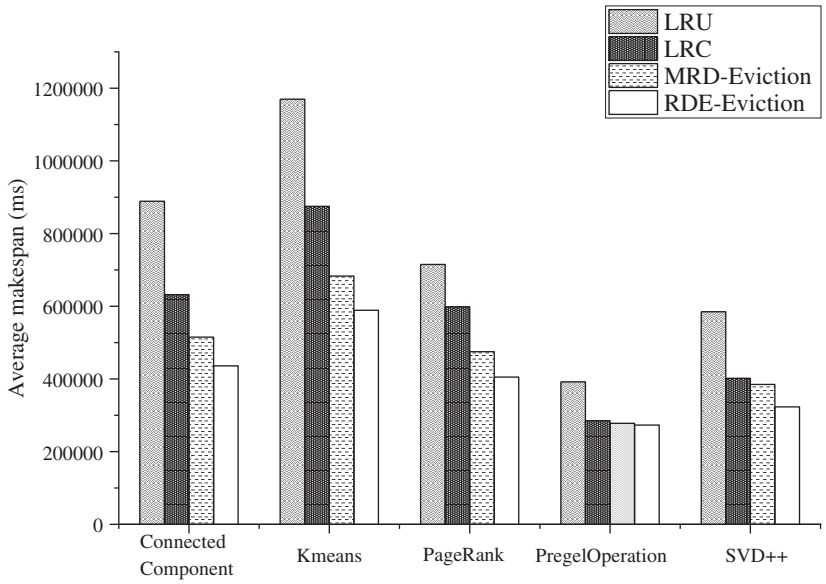

Figure 6: Overall performance compared with existing cache policies (eviction-only)

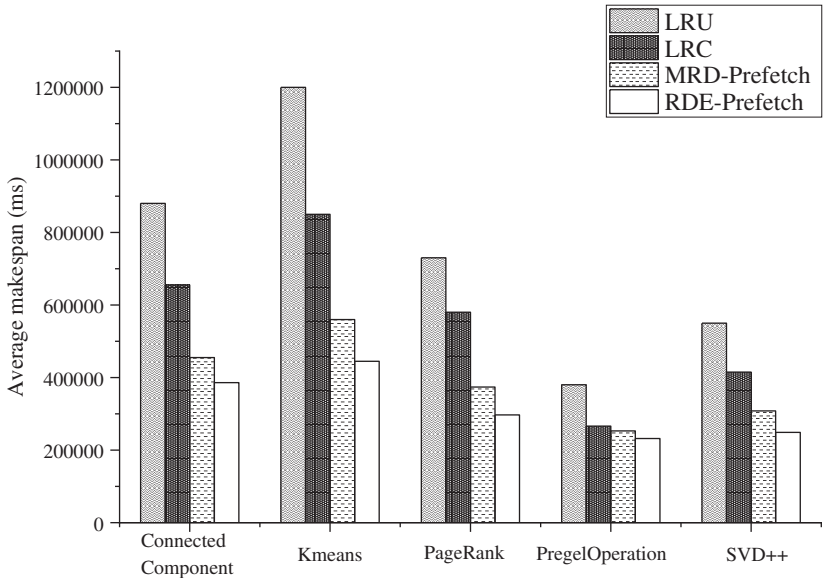

Figure 7: Overall performance compared with existing cache policies

RDE with prefetching combines the eviction policy with the prefetching method to obtain a performance. Compared to eviction-only policies, RDE improves the cache hit ratio with prefetching. It can be concluded that the performance improvement reaches 63% and 48% compared to LRU and LRC, respectively. We also compared our policy to MRD-Prefetch, which is also a DAG-based cache policy with prefetching. RDE achieves an approximately 9% to 20% advantage in performance, which benefits from the eviction policy of redundant RDDs. Especially in memory-constrained situations, RDE can take full use of memory space and make a prefetch policy easier to trigger.

4.3 Performance Comparison with Different Memory Sizes

As we mentioned above, RDE shows less memory sensitivity and performs well in memory-constrained situations. We deploy each executor memory size from 2 to 6 GB and compare RDE-Prefetch with LRU, LRC and MRD-Prefetch in several benchmarks with different configurations. Our purpose is to find the influence of the memory size on different cache policies.

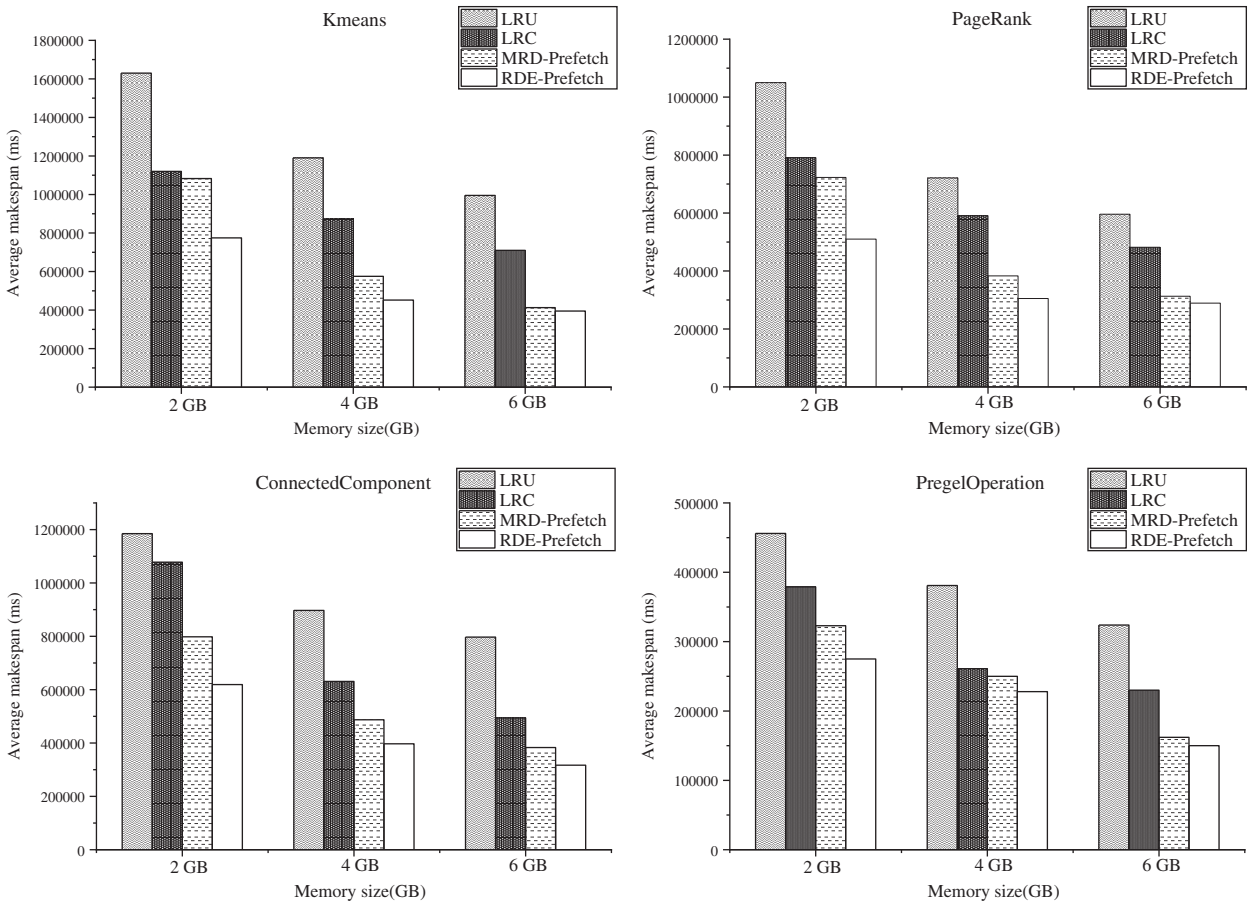

Figure 8: Performance under four cache management policies with different cache sizes

As shown in Fig. 8, each cache policy performs better as the amount of memory is increased. Policies with prefetching, such as MRD-Prefetch and RDE-Prefetch, show more advantages in clusters with more memory. However, in memory-constrained situations, such as when executors are deployed with 2 GB of memory, the cache system with MRD-Prefetch seems to be the first to encounter a memory bottleneck. The results show that with sufficient memory, MRD-Prefetch performs well with various workloads. In regard to a memory-constrained situation, MRD-Prefetch achieves a worse performance, and the application running time equals that of the system with LRC in k-means and PageRank. After analyzing the system logs, we find frequent RDD eviction occurring in memory, and there is not enough memory space to trigger the prefetch method in MRD. The prefetching method in MRD-Prefetch is limited in memory-constrained situations. RDE still outperforms in clusters deployed with low memory. It reduces the execution

time by 40% to 56% compared to LRU, which means that the prefetch method in RDE works well in memory-constrained situations. We can expect RDE to have good performance in some new fields with constrained resources, such as edge computing [22,23].

5 Conclusion

In this paper, we present a DAG-based cache management policy with redundant eviction data in Spark named RDE. RDE traverses the lineage of an application and computes the degree of dependency similarity for each RDD. Redundant RDDs have no opportunity to be cached in the workflow, which makes RDE perform better in memory-constrained situations. Moreover, we also adapt a prefetch mechanism to RDE to obtain a better cache hit ratio. Compared to the LRC and MRD policies, RDE achieves 35% and 20% improvements in performance, respectively, under memory-constrained circumstances.

Funding Statement: This work was supported by the National Natural Science Foundation of China under Grant 61100029.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. of the 2nd USENIX Workshop on Hot Topics in Cloud Computing*, Boston, MA, USA, pp. 95–101, 2010.

[2] B. Saha, H. Shah, S. Seth, G. Vijayaraghavan, A. Murthy *et al.,* "Apache tez: A unifying framework for modeling and building data processing applications," in *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, New York, NY, USA, pp. 1357–1369, 2015.

[3] R. Evans, "Apache storm, a hands on tutorial," in *Proc. of IEEE Int. Conf. on Cloud Engineering*, Tempe, AZ, USA, pp. 2, 2015.

[4] G. Ananthanarayanan, A. Ghodsi, A. Warfield, D. Borthakur, S. Kandula *et al.,* "Pacman: Coordinated memory caching for parallel jobs," in *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation*, San Jose, CA, USA, pp. 267–280, 2012.

[5] J. Wang and M. Balazinska, "Elastic memory management for cloud data analytics," in *Proc. of the USENIX Annual Technical Conf.*, Santa Clara, CA, USA, pp. 745–758, 2017.

[6] A. Nasu, K. Yoneo, M. Okita and F. Ino, "Transparent in-memory cache management in apache spark based on post-mortem analysis," in *Proc. of IEEE Int. Conf. on Big Data*, Los Angeles, CA, USA, pp. 3388–3396, 2019.

[7] G. Sun, F. H. Li and W. D. Jiang, "Brief talk about big data graph analysis and visualization," *Journal on Big Data*, vol. 1, no. 1, pp. 25–38, 2019.

[8] Y. H. Yu, W. Wang, J. Zhang, Q. Weng and K. Letaief, "OpuS: Fair and efficient cache sharing for in-memory data analytics," in *Proc. of IEEE 38th Int. Conf. on Distributed Computing Systems*, Vienna, Austria, pp. 154–164, 2018.

[9] M. Yu, R. Li and Y. Chen, "A cache replacement policy based on multi-factors for named data networking," *Computers, Materials & Continua*, vol. 65, no. 1, pp. 321–336, 2020.

[10] Vengadeswaran and Balasundaram, "An optimal data placement strategy in hadoop for data intensive applications based on cohesion relation," *Computer Systems Science and Engineering*, vol. 34, no. 1, pp. 47–60, 2019.

[11] D. Cheng, Y. Chen, X. Zhou, D. Gmach and D. Milojicic, "Adaptive scheduling of parallel jobs in spark streaming," in *Proc. of IEEE Conf. on Computer Communications*, Atlanta, GA, USA, pp. 1–9, 2017.

[12] S. Wang, Y. Zhang, L. Zhang and N. Cao, "An improved memory cache management study based on spark," *Computers, Materials & Continua*, vol. 56, no. 3, pp. 415–431, 2018.

[13] Y. H. Yu, W. Wang, J. Zhang and K. Letaief, "LRC: Dependency-aware cache management for data analytics clusters," in *Proc. of IEEE Conf. on Computer Communications*, Atlanta, GA, USA, pp. 1–9, 2017.

[14] B. Wang, J. Tang, R. Zhang, W. Ding and D. Qi, "LCRC: A dependency-aware cache management policy for spark," in *Proc. of IEEE Int. Conf. on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications*, Melbourne, Australia, pp. 956–963, 2018.

[15] T. B. Perez, X. Zhou and D. Cheng, "Reference-distance eviction and prefetching for cache management in spark," in *Proc. of the 47th Int. Conf. on Parallel Processing*, New York, NY, USA, pp. 1–10, 2018.

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

[17] M. Zaharia, M. Chowdhury, T. Das, A. Dave and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. of the 9th USENIX Symp. on Networked Systems Design and Implementation*, San Jose, CA, USA, pp. 15–28, 2012.

[18] Z. Zhao, H. Zhang, X. Geng and H. Ma, "Resource-aware cache management for in-memory data analytics frameworks," in *Proc. of IEEE Int. Conf. on Parallel & Distributed Processing with Applications, Ubiquitous Computing & Communications, Big Data & Cloud Computing, Social Computing & Networking, Sustainable Computing & Communications*, Xiamen, China, pp. 364–371, 2019.

[19] D. Niu, B. Chen, T. Cai and Z. Chen, "The classified and active caching strategy for iterative application in spark," in *Proc. of the 27th Int. Conf. on Computer Communication and Networks*, Hangzhou, Zhejiang, China, pp. 1–2, 2018.

[20] S. Wang, S. Geng, Z. Zhang, A. Ye, K. Chen *et al.,* "A dynamic memory allocation optimization mechanism based on spark," *Computers, Materials & Continua*, vol. 61, no. 2, pp. 739–757, 2019.

[21] M. Li, J. Tan, Y. Wang, L. Zhang and V. Salapura, "Sparkbench: A comprehensive benchmarking suite for in memory data analytic platform spark," in *Proc. of the 12th ACM Int. Conf. on Computing Frontiers*, Ischia, Italy, pp. 1–8, 2015.

[22] T. F. Yang, X. J. Shi, Y. Y. Li, B. B. Huang, H. Y. Xie *et al.,* "Workload allocation based on user mobility in mobile edge computing," *Journal on Big Data*, vol. 2, no. 3, pp. 105–111, 2020.

[23] Y. Xu, Z. Jin, X. Zhang and L. Zhang, "An optimization scheme for task offloading and resource allocation in vehicle edge networks," *Journal of Internet of Things*, vol. 2, no. 4, pp. 163–173, 2020.