SCIENCE CHINA
Information Sciences

LPW: An Efficient Data-Aware Cache Replacement Strategy for Apache Spark

Hui Li[1], Shuping Ji[2], Hua Zhong[2], Wei Wang[3], Lijie Xu[1], Zhen Tang[4], Jun Wei[5] and Tao Huang[2]

Citation: SCIENCE CHINA Information Sciences (2022); doi: 10.1007/s11432-021-3406-5

View online: https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

Published by the Science China Press

SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

LPW: An Efficient Data-Aware Cache Replacement Strategy for Apache Spark

Hui Li[1,2], Shuping Ji[1], Hua Zhong[1*], Wei Wang[1,2,3,4*], Lijie Xu[1,2,3,4],
Zhen Tang[1], Jun Wei[1,2] & Tao Huang[1]

[1]*State Key Lab of Computer Science, Institute of Software,Chinese Academy of Sciences, Beijing* 100190*, China;*
[2]*University of Chinese Academy of Science, Beijing* 100049*, China;*
[3]*Nanjing Institute of Software Technology, Nanjing* 210000*, China;*
[4]*University of Chinese Academy of Sciences, Nanjing* 210008*, China*

Abstract Caching is one of the most important techniques for the popular distributed big data processing framework Spark. For this big data parallel computing framework, which is designed to support various applications based on in-memory computing, it is not possible to cache every intermediate result due to the memory size limitation. The arbitrariness of *cache* API usage, the diversity of application characteristics, and the variability of memory resources constitute challenges to achieve high system execution performance. Inefficient cache replacement strategies may cause different performance problems such as long application execution time, low memory utilization, high replacement frequency, and even program execution failure resulting from out of memory. The cache replacement strategy currently adopted by Spark is the Least Recently Used (LRU) strategy. Although LRU is a classical algorithm and has been widely used, it lacks of consideration for the environment and workloads. As a result, it cannot achieve good performance under many scenarios. In this paper, we propose a novel cache replacement algorithm, Least Partition Weight (LPW). LPW takes comprehensive consideration of different factors affecting system performance, such as partition size, computational cost, and reference count. The LPW algorithm was implemented in Spark and compared against the LRU as well as other state-of-the-art mechanisms. Our detailed experiments indicate that LPW obviously outperforms its counterparts and can reduce the execution time by up to 75% under typical workloads. Furthermore, the decreasing eviction frequency also shows the LPW algorithm can generate more reasonable predictions.

Keywords Spark, memory, cache replacement, least partition weight, data-aware

Citation Hui Li, Shuping Ji, Huang Zhong, Wei Wang, et al. LPW: An Efficient Data-Aware Cache Replacement Strategy for Apache Spark. Sci China Inf Sci, for review

1 Introduction

In recent years, the entity, *big data* is widely used in various fields. Meanwhile, it brings true value to society. For example, it was reported that government departments could save more than $149 billion by using big data techniques in developed European economies, excluding the use of big data to prevent fraud and increase tax revenue [1)]. To support various large-scale data processing applications, big data analysis frameworks play a fundamental role in such applications. Many big companies invest significant resources in developing and utilizing big data analysis frameworks, such as Twitter [2)], Facebook [3)], Google [4)] and Alibaba [5)] [6)]. To adapt to different big data processing application scenarios, many big

* Corresponding author (email: zhonghua@iscas.ac.cn, wangwei@otcaix.iscas.ac.cn)

1) https://www.mckinsey.com/business-functions/mckinsey-digital/our-insights/big-data-the-next-frontier-for-innovation
2) http://bahir.apache.org/docs/spark/current/spark-streaming-twitter/
3) https://databricks.com/blog/2016/08/31/apache-spark-scale-a-60-tb-production-use-case.html
4) https://www.zdnet.com/article/google-announces-kubernetes-operator-for-apache-spark/
5) https://github.com/alibaba/SparkCube
6) https://alibaba-cloud.medium.com/setting-up-spark-on-maxcompute-171b06c7049a

data processing frameworks have been developed, including but not limited to, Apache-Hadoop [12], Tachyon [5], ApacheTez [11], Storm[7], GridGain [8], and Spark. Among them, Spark is one of the most popular frameworks due to its high performance, good scalability, and user-friendly interface. Spark is a distributed in-memory computing solution. Compared to the disk-based solutions, such as Apache Hadoop, Spark is much faster. This difference is largely because Spark can cache the intermediate calculation results in memory, and eliminate frequent disk accessing and network transmission [20]. It means the caching mechanism plays an important role in improving the performance of Spark.

Spark provides the Resilient Distributed Dataset (RDD) [21] data structure, which is a fault-tolerant collection of elements that are partitioned across a set of machines on which operations can be performed in parallel. When a programmer uses Spark to process big datasets (stored as RDD), the programmer can use the *persist* and *unpersist* application programming interface (API) to cache data in memory or release cached data from memory. However, without good knowledge of the application workloads and the available resources of the Spark cluster, a programmer may invoke these APIs in an unreasonable way. Due to the diversity of cluster resources and application characteristics, unreasonable utilization of caching would bring performance issues, such as insufficient memory resource allocation, data loss, job execution failure, frequent cache replacement, long execution time, and even worse, cluster crashing. These types of problems have often been reported in the open-source Spark community. For example, in SPARK-16440 [9], in the Word2Vec.fit() function, excessive memory consumption on the Driver will happen when hundreds of continuous training jobs are running. SPARK-14289 [10] points that a single eviction strategy might not be enough and may cause expensive time cost. In SPARK-17503 [11], caching all data easily causes memory extension and even can cause a memory leak problem. For these issues, specific solutions are proposed. For example, for the SPARK-14289 issue, one of the proposed solutions is called LCS [25]. The key idea of LCS is to use the RDD recovery time cost as the basis for replacement. This design means that when insufficient memory exists, the RDD with the lowest recovery cost will be eliminated from memory. LCS strategy can solve the SPARK-14289 issue, meanwhile, it shows better performance than the default Least Recently Used (LRU) strategy under some scenarios. However, it doesn't consider many other important factors, such as reference count, memory space occupied of partitions, etc. As a result, its generalization capacity is limited. These reported issues also suggest that even experienced programmers may not use caching in a suitable way. The need for memory varies widely during applications running, and unreasonable configurations often causes many unnecessary performance issues. Many developers are facing significant challenges to achieve high performance for critical business tasks when using Spark to process big data with given limited hardware resources. It means timely deletion of cached data from the memory could be helpful and only caching reasonable data in memory plays an important role in guaranteeing the efficiency of applications and memory utilization.

In a nutshell, Spark users usually try to cache the data in memory for re-use to speed up task execution. Under many situations, too much redundant data could be cached and the memory gradually exhausts. When the memory is not enough, the cache replacement mechanism of Spark, specifically the LRU strategy, starts to work. This process is similar to garbage collection in the Java virtual machine. However, the problem is more complex for Spark. Because Spark needs not only to collect the unused memory but also to collect the data that is potential to be reused in the future. Randomness of *cache* usage create a chaotic situation for data analysis. A challenge is that we cannot easily know which cached data is most suitable to be replaced. The LRU strategy adopted by Spark is simple and has been widely used. However, it does not work well under lots of scenarios. Let's illustrate limitations of LRU by using a simple example. In Spark, an RDD may contain a number of partitions distributed in different hosts. To boost performance, the caching mechanism of Spark works at an RDD level. Suppose we have two partitions, p_{ai} and p_{bj}, belonging to two different RDDs. They are sequentially cached in the same node. The times for computing these two partitions are one hour and five minutes, respectively. Both partitions will be used in the future. When the cache memory of that node is used up while another partition p_{ck} needs to be cached, based on the LRU algorithm, the partition p_{ai} will be evicted from cache to release memory. However, this is obviously a hasty decision considering the big computing time differences of the p_{ai} and p_{bj} partition: replacing p_{bj} is more reasonable than replacing p_{ai}. There are also scenarios

[7] http://storm.apache.org/

[8] https://www.gridgain.com/

[9] https://issues.apache.org/jira/browse/SPARK-16440

[10] https://issues.apache.org/jira/browse/SPARK-14289

[11] https://issues.apache.org/jira/browse/SPARK-17503

where we should focus on other factors, for example, when the computing times of the p_{ai} and p_{bj} are similar. So, considering different factors in a unified way is very necessary.

Table 1 Popular Cache Replacement Strategies

Cache Replacement	Frequency	Recency	Reference Count	Occupied Space	Computation Cost
Random	✗	✗	✗	✗	✗
FIFO	✗	✓	✗	✗	✗
LRU	✗	✓	✗	✗	✗
LFU	✓	✗	✗	✗	✗
LRC	✗	✗	✓	✗	✗
LPW	✓	✗	✓	✓	✓

The LRU algorithm presents limitations because it does not consider diverse factors affecting the caching performance. Based on our observation, at least four important factors need to be considered when deciding which partition should be replaced: (1) *frequency*, (2) *reference count*, (3) *occupied space* and (4) *computation cost*. Here, frequency means the period in which data would be accessed again. Reference count indicates the number of times data would be accessed. Occupied space is defined as the size of the block. Computation cost is defined as the processing time of the block. LRU only considers *recency*. A number of other widely used cache replacement algorithms, such as Random, First In First Out (FIFO), Least Frequently Used (LFU) and the recently proposed Least Reference Count (LRC) [19] algorithm, only consider a single, at most two factors. As shown in Table 1, Random doesn't consider any factor, FIFO only considers recency, LFU only considers frequency, and LRC only considers the reference count. To overcome the limitations of these existing solutions, we propose a new cache replacement algorithm, called the Least Partition Weight (LPW for short). The goal of the LPW is to minimize the total execution time of different applications. The key idea of LPW is to design a novel unified weight model to evaluate the necessity of caching for each partition. The weight model is calculated based on the analysis of various factors such as computation cost, reference count and partition dependency.

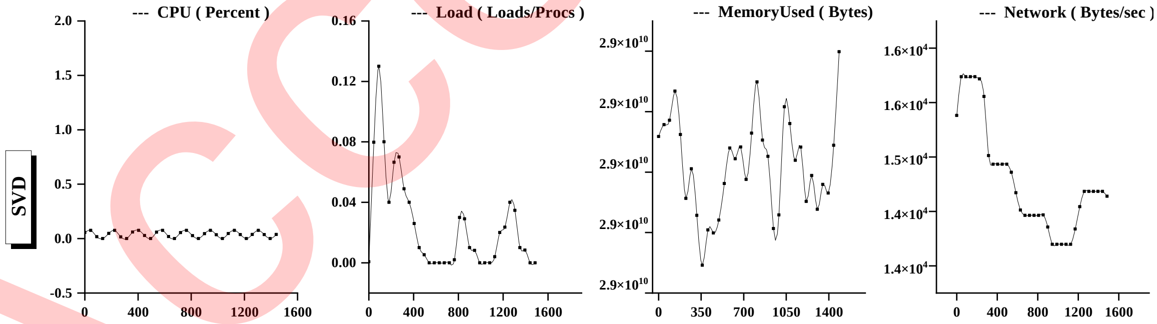


Figure 1 Execution Metrics of the SVD Application

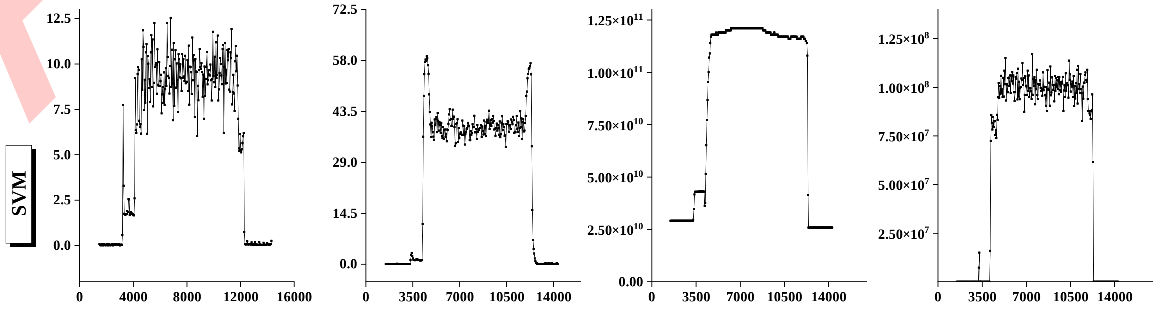


Figure 2 Execution Metrics of the SVM Application

Considering the diversity of different applications is very important. Here we use the singular value decomposition (SVD) and support vector machine (SVM) application from HiBench as the examples to show the diversity of the application workloads and the necessity to consider different factors. Figure 1 and Figure 2 show the execution metrics for SVD and SVM, respectively. The selected execution metrics included CPU, Load, MemoryUsed and Network. In this situation, CPU indicates the percentage of CPU utilization that occurred while executing at the user level. Load represented the average workload per minute (at one minute increments). MemoryUsed refers to the amount of used memory. Network represented the number of bytes transferred per second. For the SVD and SVM applications, 0.19GB and 107.3GB data were used as inputs, respectively. The Spark cluster contains four machines and is executed in the Standalone mode. The execution metrics were collected every 45 seconds. As shown in these two figures, different application workloads could exhibit very different execution behaviors. For example, compared to SVD, the CPU, Load, MemoryUsed and Network cost are higher for SVM. For SVD, the MemoryUsed footprint fluctuated during execution.

Our proposed LPW algorithm tries to consider the diversity of different applications and the potential continuous variability of memory resource requirements for a single application. LPW dynamically calculates the weight to predict the most appropriate data for making the optimal caching decision. The online replacement decision could be made according to current memory resources to optimize the cache management mechanism of Spark. Specifically, in this paper, we made the following contributions.

- A number of factors that could profoundly affect the execution efficiency of different applications on Spark, such as the cache partition data size, the number of reference count and the calculation cost were analyzed.
- Based on the analysis, the *partition weight model* and the LPW algorithm were proposed targeting to reduce the overall program execution time by selecting reasonable data to be replaced.
- The LPW algorithm was implemented in Spark and comprehensive experiments were conducted to evaluate the performance. Detailed experiments indicate that LPW could help improve the performance for a variety of applications and obviously outperform its counterparts.

The rest of this paper is organized in several sections. Section 2 reviews related background including the Spark programming model, memory management mechanism and the LRU cache replacement algorithm in Spark. Section 3 illustrates our motivation to design a new cache replacement algorithm. Section 4 proposes the *partition weight model* after analyzing different factors affecting the partition calculation. The LPW cache replacement algorithm is also presented in this section. The implementation of our solution on the Spark platform is discussed in Section 5. Comprehensive evaluation results are reported in Section 6. Section 7 surveys related work and Section 8 concludes this paper.

2 Background

In this section, we briefly present the background for caching in the Spark framework. Specifically, the programming model of Spark is discussed. After that, Spark's memory management mechanisms are described. Finally, the LRU algorithm, which is adopted by Spark for cache replacement is reviewed.

2.1 Spark Programming Model

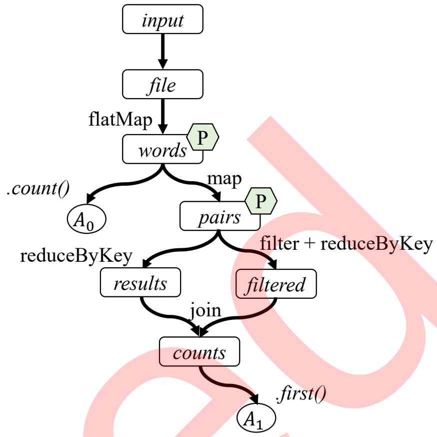
RDD is a core concept of Spark, which refers to an immutable data structure that can be divided into multiple partitions. These partitions can be computed in parallel and are stored in memory or on disk of different nodes of the cluster. Two types of operations can be executed on RDD: *transformation* (such as, *map*, *filter*, and *reduceByKey*) and *action* (such as, *count*, *first*, and *saveAsTextFile*). A *transformation* operation creates a new RDD from the existing RDDs, while an *action* operation returns the result after running some computation on an RDD. A Spark job is composed of a series of *transformation* and *action* operations. *Transformation* operations are executed in a lazy way, while *action* operations are executed in an instant way. Transformation means only when an action operation on a RDD is encountered, does the dependent transformation operations generating that RDD are kicked off.

Let us illustrate this process through a simple example as shown in Figure 3. Figure 3(a) presents a segment of source code while Figure 3(b) portrays the corresponding RDDs in addition to their dependency relationship. Since two action operations exist, there are two execution workflows. One is formed by the action A_0 in Figure 3(b) that corresponds to line 4 in Figure 3(a), and the other one is formed by the action A_1 that is corresponding to line 9. In this example, each execution workflow contains a number of *transformation* operations such as *flatMap*, *filter*, *reduceByKey* and *join*. The execution workflow triggered by A_0 can be represented as $\{input \rightarrow file \rightarrow words\}$. In Figure 3(b), the *words* RDD

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

```
1:  val file = sc.textFile(“hdfsPth…")
2:  val words =file.flatMap(x=>x.split(" "))
3:  words.persist(StorageLevel.MEMORY_ONLY)
4:  words.count()
5:  var pairs= words.map(word=>(word,1)).cache()
6:  val filtered = pairs.filter(item =>
7:      item._1.startsWith("a")).reduceByKey(_+_)
8:  val results=pairs.reduceByKey(_+_)
9:  val counts=results.join(filtered).first()
10: println("counts"+counts.toString())
11: words.unpersist()
12: pairs.unpersist()
```

(a)



(b)

Figure 3 Code Segment and Execution Graph Example

is marked as persisted since the *persist* operation is called at line 3 in Figure 3(a). The partitions of the *words* RDD are distributed at different nodes of the cluster. The execution workflow triggered by A_1 can be expressed as $\{words \rightarrow pairs;\ pairs \rightarrow results;\ pairs \rightarrow filtered;\ \ldots \rightarrow counts\}$. The *pairs* RDD is also marked as persisted since the *cache* operation is invoked at line 5 in Figure 3(a). For the second execution workflow, since the *words* RDD could be directly re-used, they persisted in the memory, and no need to recalculate it was present. After the execution of the second workflow triggered by A_1 is completed, partitions of the *pairs* RDD could also be persisted in the memory.

2.2 Spark Memory Management

A typical Spark cluster contains a master node and several worker nodes. Each worker node has an executor, which is used to manage memory and execute tasks. The memory managed by an executor can be mainly divided into two types: storage memory and execution memory. Cache-related operations, such as *persist*, *cache*, and *broadcast* consume the storage memory, while the other operations, such as *shuffle*, *join*, and *sort* will take up the execution memory. The configuration *spark.executor.memory* is used to determine the total size of an executor's available memory, while the configuration *spark.memoryFraction* is used to decide the ratio of the storage memory. For example, if the value of *spark.memoryFraction* is set at 0.5, the executor can use a maximum of 50% of the memory for caching. The remaining 50% of the memory will be used to store the dynamically created temporal objects during the execution of different tasks. *spark.memoryFraction* is an important configuration that could significantly affect the performance of the whole system. However, due to the diversity of the workloads, it is often hard to provide an ideal configuration value. An different setting might cause sever performance problems such as longer delays in executing the task. Even in the same application with a different configuration, can the performance be different. We conducted an experiment using 70GB data size as input for Bayes workload deployed in a cluster with four nodes. The result is shown in Table 2. The performance was good given a configuration *spark.memoryFraction* value of 0.6 under the executor memory value of 15GB, while performance was poor under the executor memory value of 30GB. The execution time increased by about 3 times, from 2,159 second to 7,904 seconds. By running more experiments, we found that the best performance occurred when the executor memory value is set as 30GB and the value of *spark.memoryFraction* is set as 0.1.

Table 2 Execution Time of Bayes Given Different storage.memory.fraction and Executor Memory Configurations

Spark.memoryFraction	ExecutorMemory=15GB	ExecutorMemory=30GB
0.1	8000.934	2232.523
0.3	7834.907	7929.69
0.6	2159.869	7904.824
0.9	8980.962	7895.035

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

In this experiment, we also analyze the cache replacement happened on different work nodes. Specifically, we recorded the cache replacement frequency on each worker node. The result is shown in Figure 4. The red up bound arrow indicates that replacement occurs. The blue line shows the free memory size after a block eviction. Compared to slave1, more frequent replacements occurred on the nodes (slave2 and slave3). In real-world applications, since the storage memory is often not enough to cache all intermediate computing results, frequent cache replacement may happen.

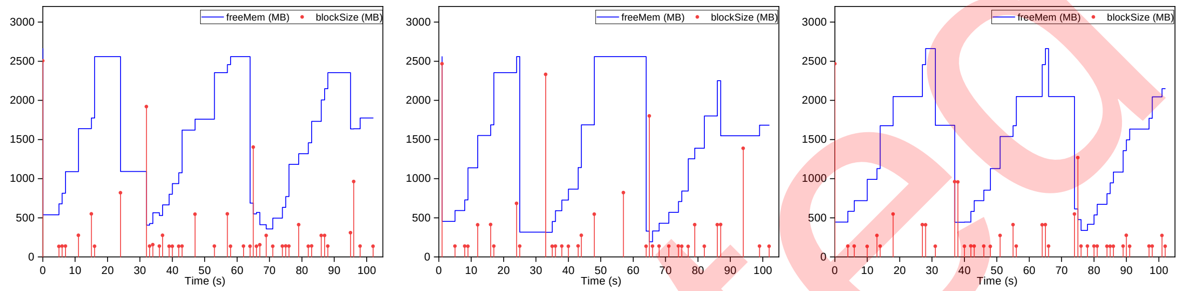


Figure 4 Free Memory and Eviction Happened in slaves : (a) Slave 1; (b) Slave 2; (c) Slave 3.

Inappropriate configuration settings can easily cause performance problems such as frequently cache replacement. To eliminate this type of problem, an effective cache replacement strategy becomes necessary and useful. In a nutshell, keeping the most useful data in memory and right evicting other data from the storage memory is one of the most important techniques for improving the performance of the whole system.

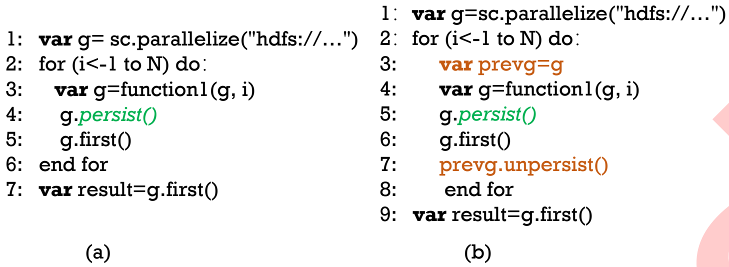
2.3 LRU in Spark

To manage the limited available memory, Apache Spark uses the LRU eviction strategy as default policy. The basic idea of LRU is to record each RDD's access times. When a new data is generated and the storage memory is not enough to cache the new data, the LRU algorithm will evict the cached partition belonging to the least recently accessed RDD. It is worth noting that the selected partition and the new partition can not belong to the same RDD. This design is to prevent partitions of the same RDD from looping in and out. LRU's mechanism of evicting the least recently used data is based on the assumption that the recently used data is more likely to be reused again. This premise is not appropriate for many applications as will be shown in the next section.

3 Motivation Examples

Challenges of Manual Cache Controlling: Figure 5 presents an iteration example, which shows the challenges of solely relying on programmers to manually control the caching and the necessity of adopting an effective cache replacement mechanism. In this example, we will analyze the limitations of two different cache-based implementations, as shown in Figure 5(a) and Figure 5(b), respectively. In Figure 5(a), the application first loads the data from HDFS and initializes the execution graph (Line 1). The iterative computation then starts, and many new execution graphs are created (Line 2 ∼ Line 6). At last, the application outputs the final result (Line 7). To improve the performance and reduce recovery time for fault tolerance, developers usually cache intermediate data into the memory by calling the *persist* API (Line 4). For each iteration, a new execution graph would be created. As the number of iterations increases, the number of caches also increases ($g_1 \rightarrow g_2 \rightarrow \ldots \rightarrow g_N$). In this case, many memory resources are wasted.

A common practice for solving this problem is to add a variable in an ad hoc manner. As shown in Figure 5(b), Spark users can assign the result of a previous calculation to the variable (*prevg* in Line 3) and cache the result of the current iteration calculation (Line 5). After the current iteration calculation is finished, *prevg* will be released from memory by calling the *unpersist* API (Line 7). The advantage of this approach is that only one intermediate calculation result is cached at any time, indicating that when the (N+1)-th job is finished, the cached result from the N-th calculation iteration will be released. This seems to be a reasonable solution. However, serious performance deterioration could happen prior to data losses. Suppose that some data loss happens because of the crash of a work node, which is very common



```
var g= sc.parallelize("hdfs://...")
for (i<-1 to N) do:
  var g=function1(g, i)
   g.persist()
   g.first()
end for
var result=g.first()
```

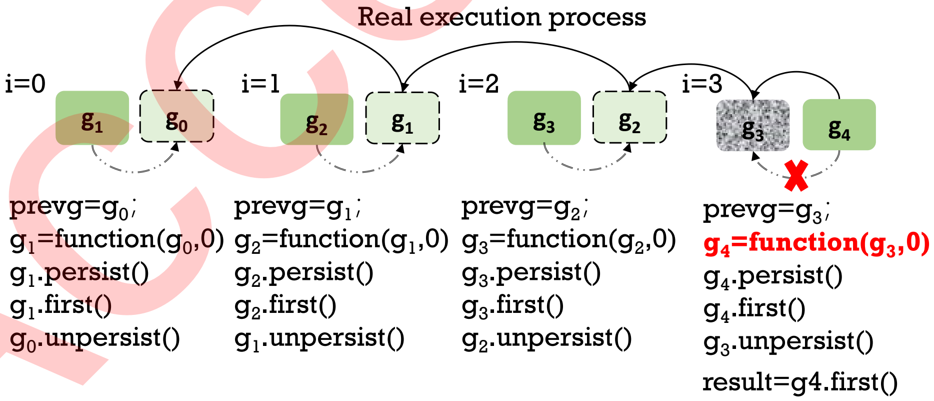
(a)

```
var g=sc.parallelize("hdfs://...")
for (i<-1 to N) do:
    var prevg=g
    var g=function1(g, i)
    g.persist()
    g.first()
    prevg.unpersist()
     end for
var result=g.first()
```

(b)

Figure 5 Code Segment for Cache API Related Example.

in large scale Spark clusters. In this case, data recalculation would be triggered according to Spark's fault tolerance mechanism. For example, suppose partial data of g_{N-1} are accidentally lost before g_N is calculated. Spark needs to recompute the data from g_0 to g_{N-1}, which could be a very time consuming process. The application may suffer from sever performance deterioration. Additional observation about recomputing the data from g_0 to g_{N-1} is shown in Figure 6. Here N is 3. When the program is running normally, the program can read g_3 directly in the memory and perform subsequent calculations while computing g_4. Suppose a buggy scenario that the program is executing the function g_4=function(g_3,0) occurs, and some partitions of g_3 are lost suddenly. The program cannot read g_3 directly in the memory since the g_3 has been lost from the memory. To perform subsequent calculations, the program needs to calculate g_3. When computing g_3 in the 2th iteration, g_2 is unpersist from the memory. So, we need to calculate g_2, continue to compute until g_0 is calculated. All in all, before computing g_4, calculation according to this order $\{g_4 \rightarrow g_3 \rightarrow g_2 \ldots \rightarrow g_0\}$ need to be performed. So, this process means this common practice is still not an ideal solution, and a good cache replacement mechanism is desired for different scenarios. Although evicting cached data according to the LRU would happen, applications may still suffer inefficient performance issues.



```
prevg=g0;
g1=function(g0,0)
g1.persist()
g1.first()
g0.unpersist()
```

```
prevg=g1;
g2=function(g1,0)
g2.persist()
g2.first()
g1.unpersist()
```

```
prevg=g2;
g3=function(g2,0)
g3.persist()
g3.first()
g2.unpersist()
```

```
prevg=g3;
g4=function(g3,0)
g4.persist()
g4.first()
g3.unpersist()
result=g4.first()
```

Figure 6 The Execution Trace of Simple Example When Some Partitions are Lost Suddenly

Necessity of Considering Different Factors: Figure 3 was used as an example to show the necessity of adopting a cache replacement strategy in addition to the necessity of considering different workload and environment related factors.

Before different features of operators in depth were observed, a simple typical test example was run, and the execution time was measured. Based on the example in Figure 3, this experiment was run in the local mode with 8GB memory and 1KB text file as input. It could be observed that the execution time of *join* operator in Stage_3 was nearly 3.7 times more than the time of *map* operator in Stage_2. The calculation time of these two operators are 70ms and 19ms respectively. In this case, *map* belonged to the operator of the Value data type, which merges the elements in each set to form a new set. Another operator *join* belonged to the Key-Value data type, which puts the same key data into a partition. Through our experiments, the features of operators were shown to be diverse. Thus, a good cache replacement should be fully aware of the features of operators, and then make appropriate replacement decisions.

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

When the memory is insufficient and blocks need to be replaced, according to the LRC, *words* will be retained because of being accessed twice, which may not necessarily improve execution efficiency. This drawback occurs because the time to calculate *words* is short and the performance bottleneck lies in the execution time of the *join* operator. Another scenario can occur in which the amount of input data is gigantic, removing the cached *words* from the memory according to the LRU still could not effectively solve the performance issues. This issues occurs because the bottleneck lies in calculation of *words*. Obviously, either LRC or LRU is difficult to guarantee applications will have a better performance in complex environments. The advantage of the LRU mechanism to keep reasonable data in memory is not obvious. In short, application execution is affected by various factors, such as computing resources, and computation time of operating on blocks. In other words, the calculation time between RDDs is diverse and the free memory is changing. As a result, predicting future data access based on single historical information (such as recency, frequency) to make a replacement decision for the big data computing framework is not effective. A good cache replacement strategy is supposed to be adaptive to various applications even though they have different characteristics.

4 Our Solution

Optimizing memory usage is one of key techniques for improving the Spark application execution performance [12] [13]. Keeping the right data in memory for re-use is an important way to improve memory utilization and reduce recalculation cost. We designed the LPW algorithm to guide Spark to choose the reasonable data for cache replacement. In this algorithm, the concept of *weight* to measure the value of caching a block was proposed. A larger *weight* indicates that the block is more valuable during the application execution and is more likely to be reused in the future. Blocks with smaller value are more likely to be evicted compared to blocks with higher weight value. Many factors could affect the execution time of Spark applications. In our design, the calculation of *weight* considered the several important factors: reference count of the block, occupied space, the calculation cost, and the historical execution information (pastmod). These factors were comprehensively considered together. Compared to current LRU algorithm adopted in Spark, which only considers the recency information of a block, our weight-based algorithm should be more feasible and effective for many different applications.

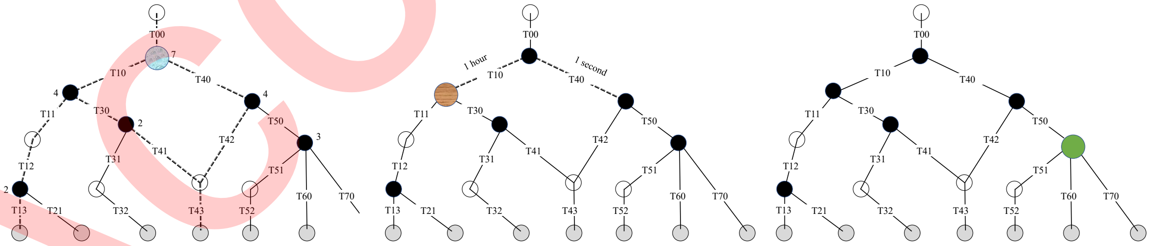


Figure 7 Cache Decisions Considering Different Factors:(a) Reference Count; (b) Computation Cost;(c) Occupied Space.

4.1 Considering Factors

Once an application is submitted to a Spark cluster, one or several jobs will be created and executed. RDD is the base components for the execution of a job. A triggered job is composed of a series of *transformation* operations on an RDD. In this paper, r_i was used to represent an RDD, and τ was used to represent a job. The execution trace of a job can be denoted as $\{r_0 \rightarrow r_1 \rightarrow r_j \, ... \rightarrow r_k\}$. Now suppose two jobs τ^p and τ^q for an application are present. If r_i is used in both τ^p and τ^q, it is better to keep r_i in memory. Figure 7 shows a sample lineage graph on which the vertices represent the RDDs. When a vertex is used by more than two jobs, it is marked in black. In this figure, the *transformation* between RDDs is connected by edges and transformation time is denoted by the symbol T. In the following sections, this figure was used as an example to explain the detailed meaning of different factors and how they can help determine which is the best cache to replace.

[12] https://unraveldata.com/common-reasons-spark-applications-slow-fail-part-1/
[13] https://unraveldata.com/common-failures-slowdowns-part-ii/

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

Reference Count. When the execution time is the same for different transformation operations, the completion time of the two jobs will be shown in Eq. (1): The first job (left most in Figure 7) $Time_1$ is counted by T_{00}, T_{10}, T_{11}, T_{12}, and T_{13}. The fourth job $Time_4$ is counted by T_{00}, T_{10}, T_{30}, T_{40},T_{41}, T_{42} and T_{43}. It is not difficult to observe that one endpoint of T_{00} is used by seven jobs and the value of the reference count is 7. Meanwhile, one endpoint of T_{50} is used by four jobs, and the value of the reference count is 4. When not enough memory exists, the best caching replacement decision is to cache the endpoint of T_{00} because of the larger Reference Count. Figure 7(a) shows the optimal caching according to the reference count.

$$\begin{aligned} Time_1 &= T_{00} + T_{10} + T_{11} + T_{12} + T_{13} \\ Time_4 &= T_{00} + T_{10} + T_{30} + T_{40} + T_{41} + T_{42} + T_{43} \end{aligned} \tag{1}$$

In short, when the reference count of the RDD is larger, therefore, it is more necessary to cache for re-use.

Computation Cost. The costs between different RDDs are often very different. The detailed execution time can be easily calculated from the task that completes the calculation of these RDDs. RDD_i and RDD_j are the different *transformation* operators. Suppose that it will take one hour to calculate RDD_i while the cost of RDD_j is only one ten minutes. In this situation, RDD_i denotes the end vertex of T_{10}. RDD_j denotes the end vertex of T_{40}. Two jobs consisted of RDD_i and RDD_j. The completion time on the partial path of two jobs are shown in Eq. (2):

$$\begin{aligned} Time_m &= RDD_i.cache() + RDD_j \\ Time_n &= RDD_i + RDD_j.cache() \end{aligned} \tag{2}$$

$RDD_i.cache()$ and $RDD_j.cache()$ denotes caching their respective data into the memory. As shown in Eq. (2), $time_m$ would be longer than $time_n$ when the calculation time of the RDD_i is larger. It is obvious that, caching RDD_i rather than caching RDD_j in memory is beneficial for improving execution efficiency. A larger computation cost implies the need for a longer calculation time. The best caching replacement decision for this example is to keep the end vertex of T_{10} because of larger computation cost. Figure 7(b) shows the optimal caching according to the computation cost. In short, the computation cost of the RDD is smaller and more likely to be evicted for releasing memory.

Occupied Space. RDDs are divided into partitions that are distributed on each node as blocks occupying memory. Based on the example in Figure 3, experiments with caching all data were performed. With an input data size of 50GB, we found that different cached partitions belonging to the *pairs* occupied different memory size. One partition is 72.1MB and another one is 1281.1MB. The larger the space occupied by the partition is, the less the necessity to cache the block is. Excessively abundant data occupy a significant portion of storage space, which may easily waste considerable memory resources and even slow down the execution efficiency. Evicting the partitions occupying a large space in memory can help decrease the application execution time. Suppose the block size of the end vertex of T_{50} is much larger than other blocks. Figure 7(c) shows the optimal caching according to the occupied space.

Pastmod. Blocks frequently used in past jobs are more likely to be accessed during subsequent job calculations. If data have been accessed more frequently during the previous period of time, this data is more likely to be used again in the future. Pastmod indicates the reference count of the partition which is computed in the completed job. For example, there are 10 jobs have been executed, and before the 11th job is executed, P_{13} is used in 5 jobs that have been executed, then the value of pastmod is calculated by 5/10 to get 0.5.

Overall, evicting the suitable data block in a timely manner when the memory is not enough is a key technique for optimizing memory usage and improving performance. As shown in the example in Figure 7, we should comprehensively consider many factors to make a good cache replacement decision.

4.2 Partition Weight Model

All factors mentioned in the above section were comprehensively considered in this study. To identify the valuable partition that needs to be kept in memory, the *weight* of a partition is calculated. For a partition $partition_i$, the formula to calculate its weight is shown in Eq. (3):

$$weight_i = \frac{cost_i * ref_i * (1 + pastmod_i)}{size_i} \tag{3}$$

$cost_i$: the execution time to calculate a $partition_i$;
ref_i : the number of jobs that depend on the current partition;
$pastmod_i$: the reference count of the partition which is computed in already finished jobs;
$size_i$: the memory space size to store the current partition;

4.3 LPW Algorithm

Our proposed LPW algorithm is designed to adapt to different Spark applications, which are executed based on in time data perception and dynamic adjustment. Algorithm 1 illustrates the pseudocode of the algorithm to identify reasonable partitions for the cached blocks in memory. The variable *partition* denotes the new block needed to be cached in memory. The variable *cachedParts* is a list of data blocks that are already cached in memory. The variable *freeMem* represents the remaining memory of the node. The variable *pWeight* denotes the weight value of each data block cached in memory. First, if the *partition* is found in *cachedParts*, the data do not need to be cached since the block is hit (Lines 1–4). The free memory and the weight in the *pWeight* (Lines 13–15) were updated. Second, the weight value of each *partition* is calculated (Line 5). The calculation method is shown in Eq. (3). During the replacement process, the weight value of *partition* is sorted in ascending order and added the sorted weight to the pQueue (Lines 6–8). Before making a decision to cache, LPW continues to check whether the space is enough when currPart is removed from memory (Lines 9–12). In the beginning, the partition from memory with the smallest weight in *pWeight* is removed and free memory space is tracked to cache *partition*. If enough space is available to cache *partition*, LPW would stop to evict block from memory and update the value of *pWeight* and *cachedParts* (Lines 13–15).

Algorithm 1 lpwRep($cachedParts, partition, freeMem$)

```
 1: if partition ∈ cachedParts then
 2:     return cachedParts[partition]
 3:     break
 4: end if
 5: weight ← Compute(partition)
 6: if freeMem < partition.size then
 7:     pQueue ← SortByWeight(cachedParts)
 8: end if
 9: while freeMem < partition.size do
10:     currPart ← pQueue.pop()
11:     freeMem + = currPart.size
12: end while
13: cachedParts.add(partition)
14: freeMem - = partition.size
15: pWeight.add(partition)
```

Algorithm 1 can be used to traverse the cached partition to find a most suitable partition to be replaced, based on the prediction of which data blocks have high probabilities to be reused. To make the algorithm more defined, we used Figure 8 as a flow chart to elaborate the execution details of the LPW algorithm.

A weight table named Weight(P_{jk}) is constructed and maintained the weight of each partition. Note that, all initial weights in the table are zero at the beginning. Suppose a new block P_{jk} that belongs to RDD_j needs to be cached. LPW would first check free memory space after which it would directly add the new partition into memory and update the weight value of the weight table if enough space is available. A new cachedParts would then haven been generated. A list of cached data blocks $R_j\{P_{j1}, P_{j2}, ...P_{jt}\}$ would also be updated to become $R_j\{P_{j1}, P_{j2}, ...P_{jt}, P_{jk}\}$. If enough space is not available to put P_{jk}, the block with the smallest weight value (not zero) would be fetched from the table and they don't belong to the same RDD. The formula can thus be expressed as shown below as P_{qw} = Minimum{Weight(P)} & Weight(P_{qw}) $\neq 0$ & $q \neq j$. The weight value of the partition is set with its minimum at zero and moved to waitingList. Spark continued to check whether enough space is available to cache P_{jk}. LPW continues to search for the smallest weight block and the value is not zero. More and more space is obtained by releasing partitions in the waitingList. In detail, after the partition P_{iw} is chosen for eviction, enough space can be finally obtained to cache P_{jk}, then partitions could be deleted from waitingList while concurrently,

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

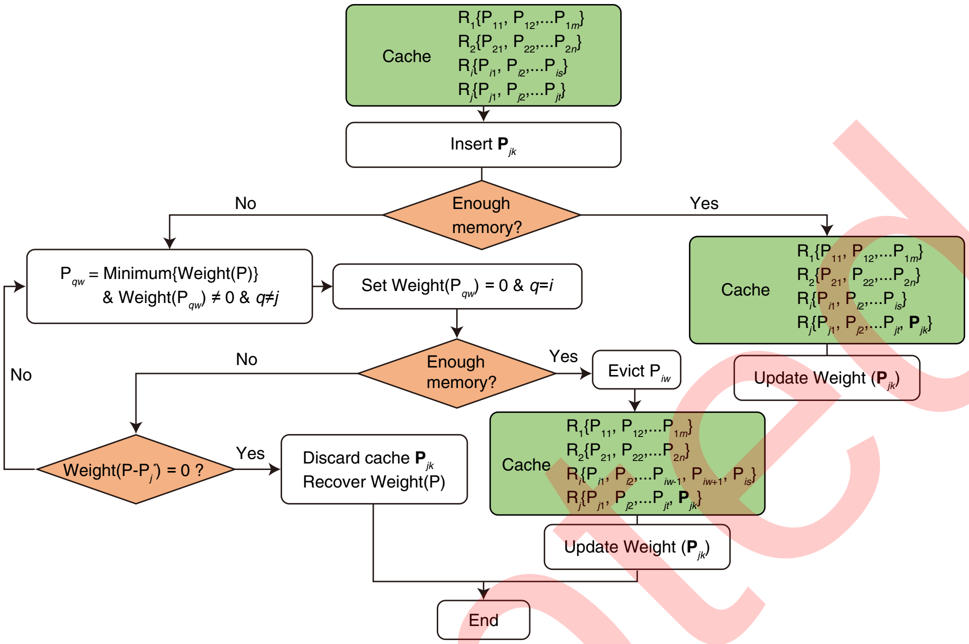


Figure 8 The Replacement Process of Cache Replacement Strategy with LPW

evicting them from the memory and updating the value of P_{qw} in the weight table. A new cachedParts are generated and a list of cached blocks $R_j\{P_{j1}$,P_{j2},...$P_{jt}\}$ is updated to $R_j\{P_{j1}$,P_{j2},...P_{jt},$P_{jk}\}$. Meanwhile, because of the eviction of P_{iw}, $R_i\{P_{i1}$,P_{i2},...$P_{is}\}$ changed to $R_i\{P_{i1}$,P_{i2},...P_{iw-1},P_{iw+1},$P_{is}\}$. Note that, when P_{iw} is replaced from the memory, it is also directly removed from the weight table. If all partitions which do not belong to the same RDD with P_{qw} are released, enough space is still not available to cache P_{jk}, then give up caching P_{jk} and recover the value of the weight table. In Figure 8, P_j^* indicates the weight value of all partitions except P_{jk}.

4.4 A Simple Example for LPW

A simplified example of a self-adaptive scenario in pseudocode is given below. The initial value of the weight table is zero. In Figure 9, five partitions$\{p_{11}$, p_{12}, p_{21}, p_{22}, $p_{23}\}$ are cached and located on the same node. Partitions p_{11} and p_{12} belongs to RDD_1, and other partitions such as p_{21}, p_{22} and p_{23} belongs to RDD_2. New data p_{13} is to be cached. The total memory size of the worker $Total_{mem}$ is initialized to 500MB.

In Figure 9(a), the memory space of p_{13} is 180MB (S_{p13}=180MB) and the cached sequence pQueue is $\{p_{11}$, p_{12}, p_{21}, p_{22}, $p_{23}\}$. The remaining memory could be easily be calculated according to the formula: $Free_{mem} = Total_{mem} - S_{R1} - S_{R2}$ to obtain enough free memory space (150MB). We compare free memory size with p_{13} space and find that not enough space for caching p_{13} data existed ($Free_{mem} < S_{p13}$). Thus, replacement process starts. First, Spark traverses the partition weight table and selects partitions where p_{13} does not belong to the same RDD. The pre-selection list $\{p_{21}$, p_{22}, $p_{23}\}$ is obtained. Second, Spark searches for the minimum weight value among the list and finally selects p_{22} as a block to be replaced. Third, the weight value of p_{22} was set to zero and move to the waitingList. The remaining memory is updated to 200MB according to formula: $Free_{mem} = Total_{mem} - S_{R1} - S_{p21} - S_{p23} + S_{p22}$. Free memory size was continuously updated with p_{13}, and it is found that enough space to cache p_{13} was available. Finally, the replacement decision according to LPW is made: p_{22} is selected for eviction from memory to cache p_{13}. We get a new cached sequence pQueue $\{p_{11}$, p_{12}, p_{13}, p_{21}, $p_{23}\}$ and the remaining memory of the node $Free_{mem}$ is changed to 20MB.

We continue to observe another situation. As shown in Figure 9(b), p_{13} still represents a new data to be cached. The memory space occupied by p_{13} is 400MB and the cache sequence pQueue is $\{p_{11}$, p_{12}, p_{21}, p_{22}, $p_{23}\}$. First, Spark checks if enough memory to store p_{13} is available, and then begins to traverse the partition weight table and find the pre-selection list $\{p_{21}$, p_{22}, $p_{23}\}$. Second, the weight value of the minimum p_{22} is set to zero and moved to the waitingList. The remaining memory is calculated usding the equation: $Free_{mem} = Total_{mem} - S_{R1} - S_{p21} - S_{p23} + S_{p22}$ to yield 200MB. In

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

this situation, not enough memory was available to cache p_{13}. We continue to choose the minimum weight value as 0.2. Partition p_{23} is chose for eviction after which the free memory size is changed to 250MB. The weight value for each partition is also updated to {0.1, 0.1, 0.5, 0.0, 0.0}. Third, p_{21} is selected sequentially from the pre-selection list $\{p_{21}\}$. Fourth, the remaining memory is computed using the equation: $Free_{mem} = Total_{mem} - S_{R1} + S_{p21} + S_{p23} + S_{p22}$, which increased to 350MB from 250MB. All partitions consist of p_{21}, p_{22} and p_{23} are both released, it is still not enough memory to cache p_{13}. Finally, the replacement decision according to LPW is made: give up caching p_{13} and reset the weight value to the initial value {0.1, 0.1, 0.5, 0.1, 0.2}.

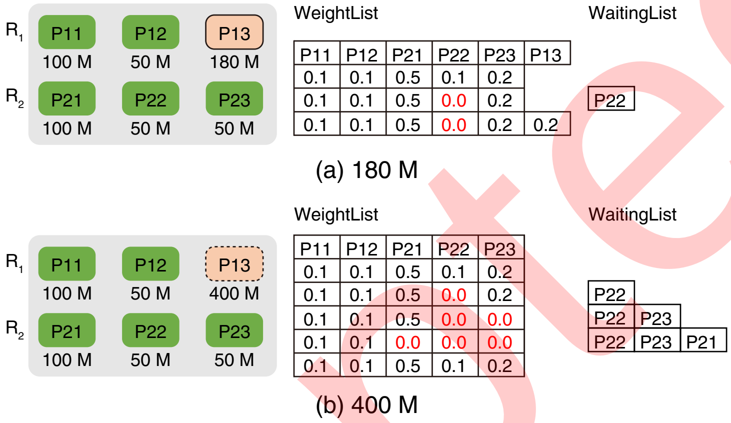


Figure 9 Cache Replacement Example of LPW

5 Implementation

LPW first collects information by modifying Spark implementation, and then calculate the weight value online on executors. After applications are submitted to Spark Cluster, DAGScheduler would divide Directed Acyclic Graph (DAG) of applications into different stages after which each stage will be divided into multiple tasks, which will be assigned to executors for execution by TaskScheduler. It is not difficult to obtain a lineage graph from the DAGScheduler, which provides an opportunity to design a data-aware cache replacement strategy that considers the characteristics of applications. By default, RDDs are divided into multiple partitions on executors of the cluster by hash partitions and then handed over to Tasks on Worker nodes. For example, partitions such as p_{01}, p_{02} of RDD_1 are in Worker1 and the rest partitions are located in Worker2. Partitions such as p_{01}, p_{12} of RDD_2 are located in Worker1 and the remaining partitions are located in Worker2. We first present system architecture that enables the implementation of the LPW in Spark and then elaborate on the implementation details. For comparison, LRC in Spark was also prototyped. Figure 10 shows an architecture overview of LPW. Our extended modules are shown in the shaded box. The Manager module is located on the Driver node, and the Monitor modules are distributed on Executor nodes.

Table 3 summarizes the key APIs for LPW. Here, the implementation details were introduced, including system architecture for the overall design and key APIS for information collection. All modules in this experiment are based on existing modules of Spark. The DAGScheduler module on the Driver node learns the dependency relationship between RDDs, allowing us to obtain the reference count (RefCount) which defines the number of times on which each RDD is relied during the calculation. RefCount is sent to WorkerManager BlockManagerSlave through BlockManagerMasterEndpoint. The BlockManagerSlave on the Worker is to keep sending heartbeats to the BlockManagerMaster to update the block (partition) information. Once the BlockManager object is created, it will also create a MemoryStore to manage the Block information. By instrumenting MemoryStore.scala, the size of the partition and the time spent in calculating could be obtained. The number of times on which each RDD is relied on in the completed job calculation is obtained and used to calculate the partition weight according to the model. The weight is calculated and saved in the worker in which the corresponding task is executed. After each job was completed and, when the dependency count information of the new job is passed to the MemoryStore, the MemoryStore will record its current dependency information such as the number of times RDD

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

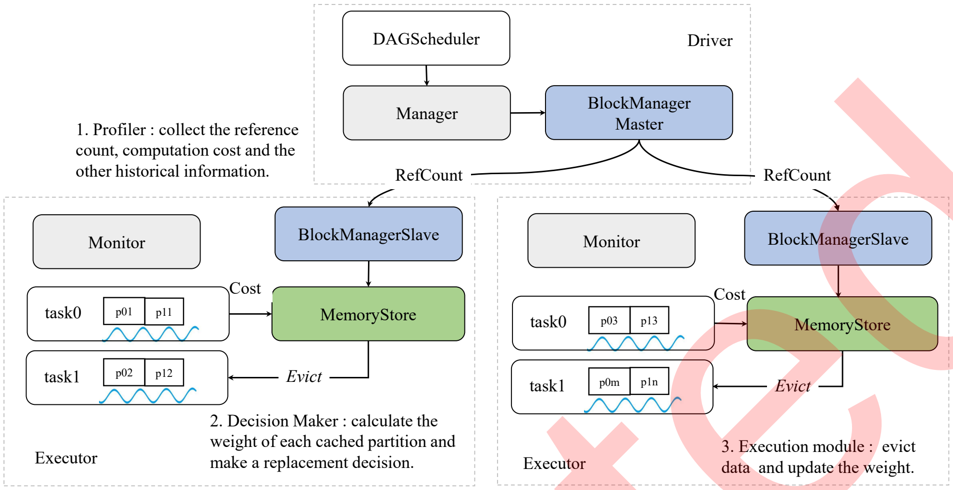

DAGScheduler
Driver
Manager
BlockManager Master
1. Profiler : collect the reference count, computation cost and the other historical information.
RefCount
RefCount
Monitor
BlockManagerSlave
task0
p01
p11
Cost
MemoryStore
task1
p02
p12
Evict
2. Decision Maker : calculate the weight of each cached partition and make a replacement decision.
Executor
Monitor
BlockManagerSlave
task0
p03
p13
Cost
MemoryStore
task1
p0m
p1n
Evict
3. Execution module : evict data and update the weight.
Executor

Figure 10 System Architecture of the LPW Solution

was referenced in the previous job and receives the new dependency information. Whenever insufficient memory on the Worker is found, the MemoryStore will send the data of the current node and select the appropriate partition to complete the replacement by using LPW decision information. An RDD with more than one partition is distributed on each node in the cluster. The life cycle of the RDD is the period from the first job to the last job that relies on the partition to complete the calculation. The computation cost of a partition contains calculation time and communication time. The ShuffleMapTask and ResultTask on the workers in the cluster record the running time of each task and the calculation cost of each partition can be obtained through code instrumentation. The BlockManager is used to manage data in Spark. We can obtain the size of memory space occupied by partitions by monitoring the API. The BlockManager on each worker makes the replacement decision based on weight and sends information to Driver.

Table 3 Key APIs for LRW Implementation

Key API	Location	Description
putOrUpdateWholeBlock	MemoryStore.scala	Update the weight value of the partition
updateRef	MemoryStore.scala	Update local RefCount
updateCost	MemoryStore.scala	Update local cost of the partition
calculateWeight	MemoryStore.scala	Calculate the weight for each partition
broadcastRef	BlockManagerMasterEndpoint.scala	Send RefCount to the BlockManagerSlave on the worker
profileRefCountStageByStage	DAGScheduler.scala	Calculate RefCount that needs to be cached in a single job.
profileRefCountOneStage	DAGScheduler.scala	Calculate RefCount that needs to be cached in a single stage.
handleJobSubmitted	DAGScheduler.scala	Calculate RefCount
calculateTrueTime	ShuffleMapTask.scala ResultTask.scala	Get the calculation time of a partition

6 Evaluation

In this section, we evaluate the performance of our proposed LPW algorithm by using the Hibench [3] [14] benchmark, which contains a number of typical application workloads. Through several experiments, we try to answer the following three questions:

RQ1: *Can LPW effectively address the performance issues faced by applications under the environment with limited memory resource?*

RQ2: *Compared to existing solutions, can LPW obviously speed up the execution of Spark applications?*

RQ3: *Does LPW introduce extra overhead during the execution of different Spark jobs?*

[14] https://github.com/Intel-bigdata/HiBench

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

In the following sections, we review the hardware and software environment used run experiments. Second, the distribution and characteristics of our adopted workloads are presented. Third, the detailed evaluation results are presented. Finally, a brief analysis of the characteristics of our proposed LPW algorithm is provided.

6.1 Environment

Our experiments are running on a Spark cluster with four machines. Each machine has 64GB of memory and an Intel(R) Xeon(R) Bronze 3106 processor with 16 cores. The CPU frequency is 1.70GHz. The disk size is 4TB, among which 1.9TB are configured as HDFS. The installed operating system is CentOS. The version of the installed Spark is 2.2.3, on which our LPW algorithm was implemented. In our experiments, the Spark's built-in LRU algorithm and the latest LRC algorithm are used as baseline to compare with our proposed LPW. In our Spark cluster, one machine is set as the master node, and the left three machines are set as slave nodes. During experiments, the tested applications are submitted to the cluster. The applications are run on the *Standalone* deploy mode. The hardware and software configuration details of our experiment environment are also shown in Table 4.

Table 4 Hardware and Software Environment

Hardware/Software configuration	Detail
CPU Model	16 * Intel(R) Xeon(R) Bronze 3106 CPU
CPU FREQUENCY	1.70GHz
RAM	64GB
DISK	4TB(1.9TB for HDFS)
OPERATING SYSTEM	CentOS Linux release 7.3.1611 (Core)
SPARK VERSION	2.2.3
SCALA VERSION	2.11.12
JDK VERSION	1.8.0_131
HADOOP VERSION	2.7.7

6.2 Workloads

To observe system performance with different cache replacement policies, our experiments were conducted with benchmarks having different characteristic workloads. In this section, the popular Hibench benchmark was chosen and system performance was evaluated. Hibench is an open-source big data benchmark tool including various application categories, such as micro, ml (machine learning), SQL, GraphX and Websearch. It can be used to evaluate the speed, throughput and system resource utilization of different big data frameworks. The sparkbench framework was chosen as our experimental framework. In total, there are 19 workloads in HiBench (V7.0). Specifically, we used two rules to select these 7 workloads for our evaluation. First, we select the workloads that are highly relied on the cache mechanism. We use the command *grep -rn '.cache('* and *grep -rn '.persist('* to search each workload which call *cache* or *persist* API. Second, for each algorithm, we also keep one. For example, /ml/DenseKmeans.scala and /dal/Densekmeans.scala both implemented the KMeans algorithm, we only choose the first one in our experiments. To differentiate from these applications, seven typical applications are finally obtained from HiBench Benchmark as our experimental objects.

HiBench supports the generation of input data with different scales, such as tiny, huge, large and gigantic-scale. To simulate big data scenarios as much as possible, larger and above data scale as input are selected. We calculate the input data size for each workload, as shown in the fourth column of Table 5. More details about Hibench are elaborated in Table 5. For the seven workloads (C1–C7), we found *cache* or *persist* API was called. The column 3 in Table 5 (category) shows the application category. It can be seen that *cache* or *persist* API is widely used in iterative scenarios, such as Machine Learning and GraphX. The last column shows some other important input parameter configurations required for applications to run.

6.3 Evaluation Results

For each workload, these workloads in cluster configured with different executor memory and driver memory were deployed. Applications were run many times in the Standalone mode, and then use the

Table 5 Summary of workloads for Hibench

ID	Category	Workload	Input Data Size (GB)	Parameter Configurations
C1	Websearch	Pagerank	2.79	pages=5000000; num_iterations=3
C2	GraphX	Nweight	0.29	edges=10000000; degree=3; max_out_edges=30
C3	Machine Learning	Kmeans	37.41	dimensions=20; num_of_samples=200000000
C4	Machine Learning	LogisticRegression	37.25	examples=10000; features=100000
C5	Machine Learning	PCA	0.22	examples=1000; features=1000
C6	Machine Learning	SVM	107.3	examples=120000; features=300000
C7	Machine Learning	Bayes	70.14	pages=20000000; classed=20000; ngrams=2

average execution time. The system performance of each workload are observed and tested under different cache replacement strategies. The other configuration parameters were the default value, such as MemoryFraction, which was set to 0.6. Execution time, speed up and eviction frequency under different cache replacement strategies were compared.

Application Execution Time. A set of experimental results under the configuration of 15GB executor and driver memory were summarized. First, execution time of each application under different cache replacement policies, i.e., LRU, LRC, and LPW were observed. The overview result is shown in Figure 11.

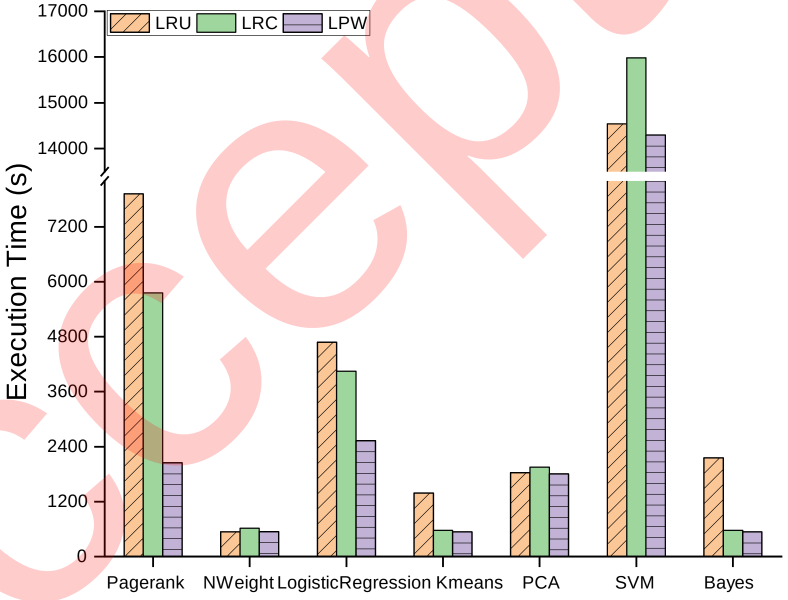


Figure 11 Execution Time for Different Workloads with 15GB of Executor Memory and Driver Memory

As we all know, the configuration of cluster parameters has a great impact on the performance of the application. The execution time of workloads against different *spark.memoryFraction* (from 0.1 to 0.9) was measured, and results are depicted in Figure 12. To answer RQ1, LPW effectively addressed the performance issues faced by applications, especially in limited resource. Compared with LRC, application Nweight have poor performance especially in smaller spark.memoryFraction, such as 0.1 and 0.3 as shown in Figure 12(a). This is because the nweight has only 1 job. When the only one job is executed, the advantage of reusing cached data is not obvious. However, when the LogisticRegression application with 44 jobs and Bayes with 9 jobs are executed, the cached data is relied on by many jobs. The performance has been greatly improved.

SpeedUp. Next, speedup and average execution time of all workloads having *cache* or *persist* API in HiBench test suites in-depth were observed. The experimental results are shown in Table 6. It can be seen that Kmeans, logistic Regression and Bayes show good performance under the configuration of 15GB executor memory and driver memory. The other parameter configuration of the experimental results is default. This technique saves more than 45.80% time over LRU. The performance improvement of Bayes is the most noticeable, which can be up to 74.87% higher than LRU. Caching reasonable data reduces the execution time for some typical applications. It should be noted that some applications in HiBench have

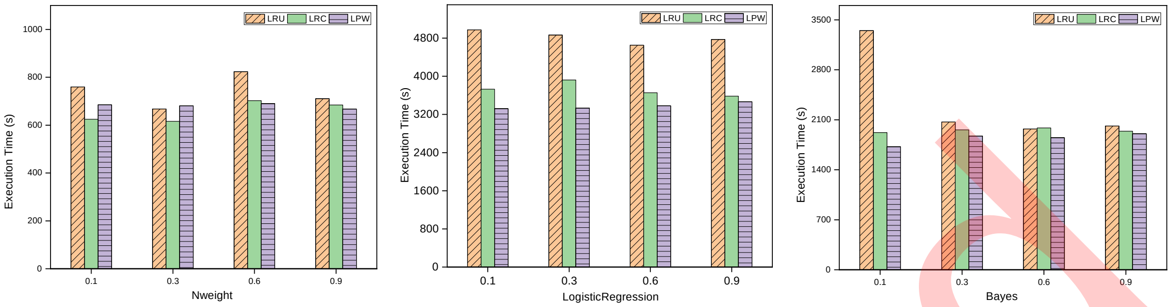


Figure 12 Execution Time under the LPW, LRU and LRC (a) Nweight ; (b) LogisticRegression; (c) Bayes.

good improvement because of the memory-intense feature. It is common for iterative scenarios such as Kmeans and LogisticRegression to cache input data in memory before model training. These distributed applications repeatedly perform the same calculation on different data. For answering RQ2, LPW can effectively speed up the execution of Spark applications compared to LRU and LRC.

Table 6 Execution time and speedup with different cache replacement strategy under 15GB Executor Memory

Workload	LRU (s)	LRC (s)	LPW (s)	SpeedUp-LRC	SpeedUp-LRU
Pagerank	7921.324	5760.366	2047.133	27.28%	74.16%
Nweight	541.5135	618.2463	543.3262	-14.17%	-0.33%
Kmeans	1385.608	570.9917	541.4555	58.79%	60.92%
LogisticRegression	4677.865	4044.374	2535.377	13.54%	45.80%
PCA	1829.108	1953.538	1805.344	-6.80%	1.30%
SVM	14540.98	15981.48	14299.84	-9.91%	1.66%
Bayes	2154.762	570.9917	541.4555	73.50%	74.87%

To verify the effectiveness of our cache replacement algorithm in-depth, we focused on selecting workload having *cache* API. The seven workloads of Hibench were chosen. Figure 13 sketched out the results. To answer RQ3, LPW is lightweight without bringing extra overhead for the performance of Spark applications without *cache* or *persist* API. For these seven different characteristics of workloads, we can see that when, compared with LRU, these applications have better performance. Especially, in the case of LogisticRegression(LR) and Bayes, they have excellent performance. To answer RQ2, LPW can speed up the execution of Spark applications compared to the LRU and the other state of solutions such as LRC.

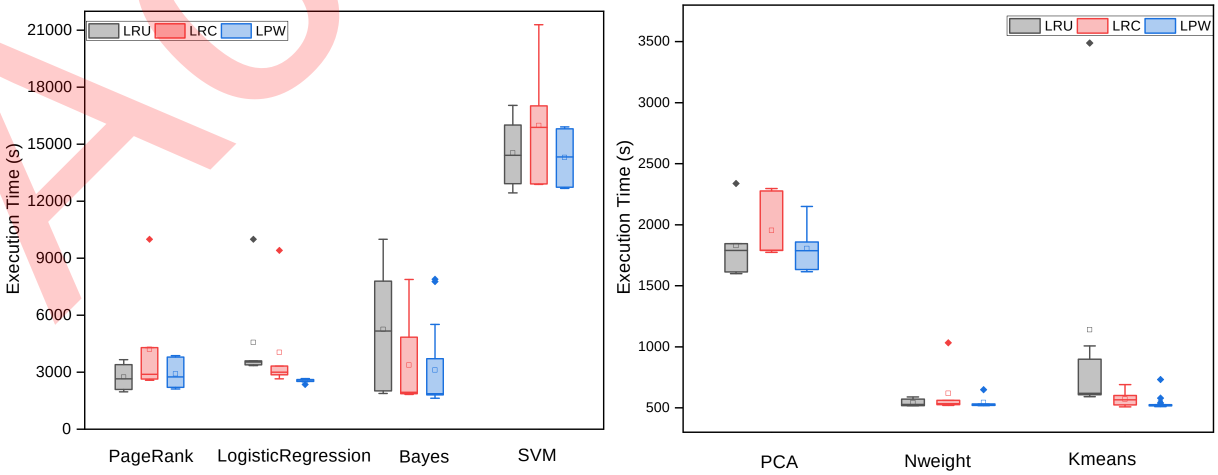


Figure 13 Execution Time for Workloads with Different Cache Replacement Policies

Eviction Frequency. The efficiency of LPW can also be illustrated from evicting frequency. Less eviction frequency for the same application under the same configuration exists. For example, the number of evictions for Bayes workload under different cache replacement policies was further summarized. As shown in Table 7, during the entire application execution process, the total number of replacements

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

that occurred decreased. LRU had 3958 evictions and LRC had 3718 evictions, while LPW had 3685 replacements. Additionally, the frequency of replacement on each node was also reduced. For Node_1, the number of evictions was 1376 under LRU, while the replacement number under LRU was 1185, indicating that our method LPW could effectively predict hot spot data to cache without causing frequent replacement.

Table 7 Eviction Numbers of Bayes with Different Cache Replacement Policies

Cluster-NodeID	LRU	LRC	LPW
Node_1	1376	1227	1185
Node_2	1391	1346	1376
Node_3	1191	1145	1124
Total	3958	3718	3685

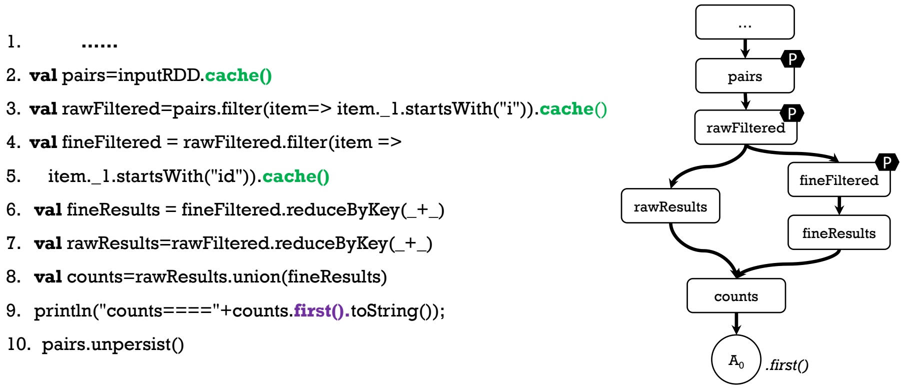
Compared with LRU and LRC, LRW presents good advantages in workloads such as graph computing and machine learning. The main reason is that these applications call the *cache* or *persist* API, when the optimal partitions are selected for replacement.

6.4 Analysis and Discussion

Compared with existing cache replacement algorithms, the extra overhead introduced by our proposed LPW algorithm was quite small. Only the newly introduced factors were collected and maintained. The calculation time for these factors was negligible. For example, when a job is executed, the time to compute a partition and the space needed to store that partition can be gotten easily. For the reference count factor, some network overhead is needed, since the reference count of an RDD needs to be synced from the master to the slaves. However, this overhead is also very small since this reference count information only has several bytes of data. These factors for different partitions are maintained in an in-memory table. The formula to calculate weight of different partitions is simple, and the consumed time is negligible.

We selected typical and popular open-source Spark applications in Github. Experiments show that LPW is promising in identifying hot data for Spark applications. A challenge in efficient cache replacement is to select important factors affecting performance. Our studied factors, a.k.a., computation cost, reference count and block size, are representative and important. The arbitrariness of *cache* API usage, the diversity of application characteristics, and the variability of memory resources construct challenges for efficient system performance. LPW can dynamically be aware of hot data and provide an wise decision to improve performance for various workloads.

An independent evaluation to evaluate LPW is provided as shown in Figure 14. There were three RDDs that would be cached. Given the same input data and the same workload, parameters such as executor memory were configured the same. The workload was run in local[8] mode and the memory with -Xmx=512MB. We then observed the execution time and detailed replacement process under the LPW and LRU. In our experiments, the execution time of the example were 79s and 96s under LPW and LRU, respectively.



```
      ......
val pairs=inputRDD.cache()
val rawFiltered=pairs.filter(item=> item._1.startsWith("i")).cache()
val fineFiltered = rawFiltered.filter(item =>
  item._1.startsWith("id")).cache()
 val fineResults = fineFiltered.reduceByKey(_+_)
 val rawResults=rawFiltered.reduceByKey(_+_)
 val counts=rawResults.union(fineResults)
 println("counts===="+counts.first().toString());
 pairs.unpersist()
```

Figure 14 Code Segment and Corresponding Lineage Graph

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5

Logs were collected and analyzed under different cache replacement strategies. Our results are shown in Figure 15. Under the guidance of the LRU strategy, the cache blocks belonging to *paris* (RDD3) and *rawFiltered* (RDD4) were preferentially eliminated. On the contrary, the decision made by LPW was to give priority to removing cache blocks belonging to *fineFiltered* (RDD5). In calculation, when compared with RDD5, RDD3 and RDD4 were accessed first . According to recency in LRU, blocks of RDD3 and RDD4 were firstly evicted as shown in Figure 15 (a). According to LPW, blocks in RDD5 with the smallest weight were first eliminated under the LPW as shown in Figure 15 (b). When the application is executed, evicting RDD3 or RDD4 would not be the best choice. Because of the calculation time and other factors, the weight value of these two RDDs calculated under LPW would be greater than the weight of RDD5. The optimal choice would be to evict partitions of RDD5. Our experiments showed that LPW can guide Spark to keep the reasonable data to be reused and evict other data for replacement to enhance execution efficiency.

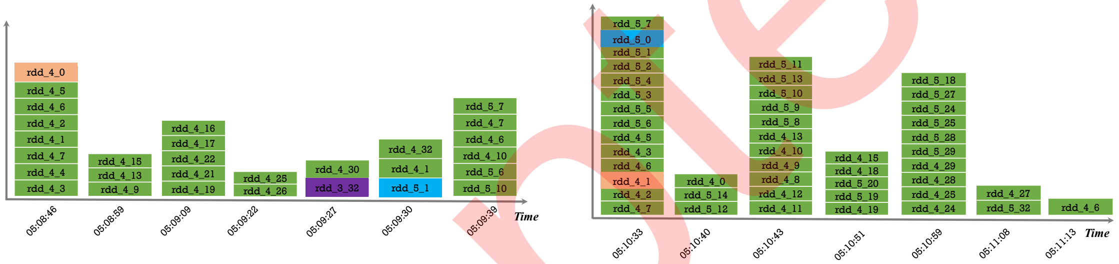


Figure 15 Cache Replacement Process under LRU and LPW (a) Replacement under LRU ; (b) Replacement Process under LPW.

7 Related Works

Cache Replacement Strategies: Remarkable advancements in identifying factors affecting caching are ongoing. Many cache replacement strategies, including traditional strategies, such as LFU, FIFO, and the newer work LRC [19] can be found. GD-Wheel [23], a cost-aware replacement policy can reduce the total recomputation cost and average latency for web applications. An imitation learning approach PARROT [24] has been proposed, which automatically learns data access patterns by leveraging Belady and computing the optimal eviction decision. The LRFU [4] algorithm takes the computational cost and size of the data block into account but does not consider the number of uses of the data block. The AWRP [13] algorithm captures the frequency of data usage, but it misses the size of the data block. LERC [18] stores the relevant blocks of the computing task as a whole and keeps them in the memory thus, omitting the data dependency. The LRC algorithm puts the concept of the number of references forward. The system monitors the number of references of each data block in the memory of each slave node. The principle of cache replacement is to remove the data block from the memory with the lowest number of references. The performance of LRC is better than LRU because it considers the problem of data dependency when the program is running. LRC evicts the data that will not be used in subsequent calculations in time and retains the data that will be used multiple times in the future not considering the data calculation. For the Spark framework, these algorithms ignore some important factors such as the data dependency or the computational cost of data. It is often difficult to have better performance in various scenarios just based on frequency or recency.

Memory Optimization Solutions: in the data parallel computing framework, caching data is one of the key technologies for achieving performance improvement. Dache [22] proposed caching intermediate result to reuse in MapReduce framework. For the Spark framework, Yang [17] showed that the technology backs up intermediate data in different storage levels for re-use without recalculation to achieve higher execution efficiency. Many research studies have elaborated performance issues caused by persisted data in memory during data processing. As developers understand the difference in professional knowledge such as caching mechanism, it brings uncertainty in caching API usage, which further causes performance problems for the application. In real-world cases, Spark's typical and representative built-in libraries such as GraphX and MLlib currently generally have improper use of cache API codes that cause application performance problems. Memory cache optimization aims to improve the utilization of data cached in memory during data processing and optimize application execution. Xu [15] proposed one of the most common reasons for an executor out of memory (OOM) is that developers always want to cache or load too much information into memory. Kay Ousterhout [9] pointed out that the increase in bandwidth and the use of SSDs or disk arrays can effectively solve network or disk IO problems. Memory-based computing frameworks such as Spark have performance bottlenecks mainly in CPU and memory rather than network overhead or Disk IO and other issues. It is of great significance to improve memory utilization in big data parallel data processing systems. The MemTune [16] considers the partial data dependency of the currently running task. In the multi-tenancy scenario. MemTune traverses all submitted DAG graphs after each computing task is completed, for which the complexity is relatively large. Stark [7] optimized memory calculations on dynamic data set collections and enforced data locality to avoid unnecessary data duplication and shuffling, ignoring whether the data is needed for subsequent calculations. PACMan [1] designed two cache replacement strategies to coordinate access to distributed caches. The first one cleared a large amount of incomplete inputs to minimize the average completion time. Another one is to clear infrequently accessed inputs to maximize cluster efficiency. Data access dependencies about applications are not considered. The work [10] was conducted on the analysis of the temporal and spatial distribution of information such as data references in the entire workflow. Neutrino [14] captured the data access dependencies between RDDs in different stages of the Spark application, and adaptively converted or discarded RDDs from different cache storage levels. If the cost of data calculation is added, it is more meaningful to make appropriate caching decisions for the entire cluster.

8 Conclusion

Spark is a widely used distributed computing framework for processing big data. Intermediate result caching is an important mechanism for Spark to achieve high performance. Due to the limitation of storage memory, the cached data often need to be replaced. As a result, cache replacement strategies can largely affect the performance of Spark. However, the LRU cache replacement algorithm, currently adopted by Spark, only considers the least recently accessed and simply evict the least recently used data. This mechanism presents limitations for many applications. To overcome the limitations of LRU and further improve the performance of Spark, we designed a data-aware cache replacement algorithm, the LPW. In view of the diversity of application characteristics and the continuous variability of memory usage requirements, the LPW establishes a unified weight model based on factors and calculates the tracked statistics for the block to identify hot data to be swapped into the memory. The weight is used as the basis for a replacement decision. Our comprehensive experiments show the effectiveness of LPW especially for iterative applications. LPW outperformed related algorithms, such as LRU and LRC, with respect to the execution time and eviction frequency.

Acknowledgements This work was supported by National Natural Science Foundation of China (Grant Nos. U20A6003, 61802377 and 61872340) and Youth Innovation Promotion Association at CAS.

References

1 Ananthanarayanan G, Ghodsi A, Warfield A, et al. Pacman: Coordinated memory caching for parallel jobs. In: Proceedings of Symposium on Networked Systems Design and Implementation, San Jose, CA, 2012. 267-280.

2 Gonzalez J E, Xin R S, Dave A, et al. GraphX: Graph processing in a distributed dataflow framework. In: Proceedings of USENIX Symposium on Operating Systems Design andImplementation, Broomfield, CO, 2014. 599–613.

3 Huang S S, Huang J, Dai J Q, et al. The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In: proceedings of International Conference on Data Engineering Workshop, Long Beach, CA, 2010. 41-51.

4 Lee D H, Choi J, Lim J H et al. Lrfu: A spectrum of policies that subsumes the least recently used and least frequently used policies. Computers, 2001, 12:1352-1361.

5 Li H Y, Ghodsi A, Zaharia M et al. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In: Proceedings of the ACM Symposium on Cloud Computing, Seattle, WA, 2014. 1-15.

6 Li H, Wang D, Huang T Z, et al. Detecting cache-related bugs in spark applications. In: Proceedings of International Symposium on SoftwareTesting and Analysis,Virtual Event, USA, 2020. 363-375.

7 Li S, Amin M T, Ganti R et al. Stark: Opti-mizing in-memory computing for dynamic dataset collections. In: Proceedings of International Conference on DistributedComputing System,Atlanta, GA, 2017. 103-114.

8 Meng X R, Bradley J, Xin D et al. MLlib: Machine learning in Apache Spark. Machine Learning Research, 2016, 17(34):1–7.

9 Ousterhout K, Rasti R, Ratnasamy S et al. Making sense of performance in dataanalytics frameworks. In: Proceedings of USENIX Symposium on Networked Systems Design and implementation, Oakland, CA, 2015. 293–307

10 Perez T B, Zhou X B, Chen D Z. Reference-distance eviction and prefetching for cache management in spark. In: Proceedings of International Conference on Parallel Processing, Eugene, OR, 2018. 1-10

11 Saha B, Shah H, Seth S et al. Apache tez: A unifying framework for modeling and building data processing applications. In: Proceedings of International Conference onManagement of Data, Melbourne, Victoria, 2015. 1357–1369

12 Shvachko K, Kuang H, Radia S et al. The hadoop distributed file system. In: Proceedings of Symposium on Mass Storage Systems and Technologies, 2010. 1-10

13 Swain D, Paikaray B, Swain D. Awrp: adaptive weight ranking policy for improving cache performance, Computer Science, 2011, 3: 2151-9617

14 Xu E, Saxena M,Chiu L,Neutrino: Revisiting memory caching for iterative data analytics. In: Proceedings of USENIX Workshop on Hot Topics in Storage and File Systems, Denver, CO, 2016. 16–20

15 Xu L J, Dou W S, Zhu F et al. Characterizing and diagnosing out of memory errors in mapreduce applications. Journal of Systems and Software, 2018, 137: 399-414

16 XU L, Li M, Zhang L, et al. Memtune: Dynamic memory management for in-memory data analytic platforms. In: Proceedings of International Parallel and Distributed Processing Symposium, Chicago, IL, 2016. 383-392

17 Yang Z Y, Jia D L, Ioannidis S, et al. Intermediate data caching optimization for multi-stage and parallel big data frameworks. In: Proceedings of International Conference on Cloud Computing, San Francisco, CA, 2018. 277-284

18 Yu Y H, Wang W, Zhang J, et al. Lerc: coordinated cache management for data-parallel systems. In: Proceedings of Global Communications Conference, 2017. 1-6

19 Yu Y H, Wang W, Zhang J, et al. Lrc: Dependency-aware cache management for data analytics clusters, In: Proceedings of reference on Computer Communications, Atlanta, GA, 2017. 1-9

20 Zaharia M, Chowdhury M, Das T, et al. Fast and interactive analytics over hadoop data with spark. The Advanced Computing Systems Association, 2012, 37: 45-51

21 Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In: Proceedings of USENIX Conference on Networked Systems Design and Implementation, 2012.

22 Zhao Y X, Wu J, Liu C. Dache:A data aware caching for Big-Data applications using the MapReduce framework. Tsinghua Science and Technology, 2014, 19:39-50.

23 Li C, Cox A L. GD-Wheel:A cost-aware replacement policy for key-value stores. In: Proceedings of the European Conference on Computer Systems, France, Bordeaux, 2015.1-15

24 Liu E, Hashemi M, Swersky K, et al. An imitation learning approach for cache replacement. In: Proceedings of the International Conference on Machine Learning. 2020.6237-6247.

25 Geng Y Zand Shi X H and Pei C, et al. LCS: An Efficient Data Eviction Strategy for Spark. 2017,45(6):1285–1297.

https://engine.scichina.com/doi/10.1007/s11432-021-3406-5