Computers, Materials & Continua

Tech Science Press

DOI: 10.32604/cmc.2024.046807

ARTICLE

Research on Performance Optimization of Spark Distributed Computing Platform

Qinlu He[1,*], Fan Zhang[1], Genqing Bian[1], Weiqi Zhang[1] and Zhen Li[2]

[1]School of Information and Control Engineering, Xi'an University of Architecture and Technology, Xi'an, 710054, China

[2]Shaanxi Institute of Metrology Science, Xi'an, 710043, China

*Corresponding Author: Qinlu He. Email: luluhe8848@hotmail.com

Received: 15 October 2023 Accepted: 07 April 2024 Published: 15 May 2024

ABSTRACT

Spark, a distributed computing platform, has rapidly developed in the field of big data. Its in-memory computing feature reduces disk read overhead and shortens data processing time, making it have broad application prospects in large-scale computing applications such as machine learning and image processing. However, the performance of the Spark platform still needs to be improved. When a large number of tasks are processed simultaneously, Spark's cache replacement mechanism cannot identify high-value data partitions, resulting in memory resources not being fully utilized and affecting the performance of the Spark platform. To address the problem that Spark's default cache replacement algorithm cannot accurately evaluate high-value data partitions, firstly the weight influence factors of data partitions are modeled and evaluated. Then, based on this weighted model, a cache replacement algorithm based on dynamic weighted data value is proposed, which takes into account hit rate and data difference. Better integration and usage strategies are implemented based on LRU (Least Recently Used). The weight update algorithm updates the weight value when the data partition information changes, accurately measuring the importance of the partition in the current job; the cache removal algorithm clears partitions without useful values in the cache to release memory resources; the weight replacement algorithm combines partition weights and partition information to replace RDD partitions when memory remaining space is insufficient. Finally, by setting up a Spark cluster environment, the algorithm proposed in this paper is experimentally verified. Experiments have shown that this algorithm can effectively improve cache hit rate, enhance the performance of the platform, and reduce job execution time by 7.61% compared to existing improved algorithms.

KEYWORDS

Spark; memory optimization; memory replacement strategy

1 Introduction

With the trend of digital transformation in various industries, the global data volume is exploding, expected to reach 175 ZB by 2025 [1]. In recent years, the availability of data has continued to improve in terms of quantity, speed, variety, and accuracy, ushering in a new era of big data analysis. Big data is widely used in various industries. It is expected that the market size of big data and business analysis will reach $298.3 billion by 2024 [2].

Big data analysis requires timely processing of data, and its processing methods differ greatly from traditional models. The processing method has changed from computational processing to data processing, which makes traditional processing models adapt to the requirements of big data environments. The scale of big data has long exceeded the ability of a single machine. Therefore, Google proposed the idea of distributed storage and parallel computing [3], aiming to break down big data into manageable sizes, process data on distributed clusters, and then make the data available for users to use. On this concept, the distributed computing platform Hadoop [4] was born, with its core being the MapReduce [5] parallel computing framework. Its principle is "task division and result aggregation". When calculating jobs, MapReduce continuously reads and writes intermediate data to the distributed storage system, but this generates a large amount of I/O overhead and affects the job execution time. Spark proposes an elastic distributed dataset [6] based on Hadoop, which stores intermediate results in memory and achieves in-memory computing, avoiding a large amount of disk I/O overhead and reducing data processing time. For iterative jobs, the Spark platform is more than ten times faster than the Hadoop platform due to its in-memory computing feature [7]. Because in-memory computing has enormous advantages in processing massive data, Spark has broad application prospects in large-scale computing applications such as machine learning, image processing, etc.

Many domestic and foreign enterprises have applied the Spark platform in real production environments, and some enterprises have Spark clusters with a scale of thousands of nodes and data processing scales reaching hundreds of millions. Such large-scale clusters can generate expensive operational expenses, and efficient utilization of computing clusters is important for enterprises. Even a small improvement in utilization rate can save a lot of costs [8]. How to improve the performance of Spark has become a hot research direction. The core of Spark is parallel computing and in-memory computing. In the Spark platform, the module closely related to parallel computing is the task scheduling module, which is responsible for assigning tasks to nodes in the cluster to achieve parallel computing on the cluster. In-memory computing intermediate results are stored in memory, reducing disk read overhead or re-calculation overhead. The memory resources of nodes are limited. When the memory space is full, it is necessary to store higher-value data partitions according to the cache mechanism to improve the cache hit rate. Therefore, task scheduling and cache replacement are particularly important for Spark performance optimization. In practical application scenarios, due to reasons such as cluster upgrading and expansion, the node configuration changes, and the cluster is prone to heterogeneity. However, the current Spark platform has not made targeted adjustments for heterogeneous clusters and still assigns tasks based on ideal conditions for homogeneous clusters [9]. The Least Recently Used (LRU) algorithm only considers the time of the most recent access when selecting the RDD to be replaced, and if RDDs with high reuse frequency and high re-computation cost are evicted from the memory cache area, subsequent reuse will bring significant re-computation overhead and slow down the overall task execution progress [10,11]. Therefore, further research is needed to improve the cache replacement algorithm in Spark to achieve better performance and efficiency.

This paper mainly analyzes and studies the overall architecture and workflow of Spark, analyzes the workflow and memory management of Spark, proposes optimizations for Spark's cache replacement strategy, and optimizes other parts of its memory according to Spark's workflow.

(1) The architecture, workflow, memory management, and caching strategy of the Spark system are analyzed, and this paper is used as the theoretical basis for the optimization of the cache replacement strategy.

(2) A cache replacement strategy based on data value is proposed, and a better combination and use strategy is carried out based on LRU considering the hit rate and the difference of data.

(3) The storage location and read recovery of RDD caches have been optimized accordingly, a new persistence level has been proposed, and Spark security and fault tolerance have been optimized.

(4) The results of experiments under different strategies are given. The results are compared and analyzed to draw corresponding conclusions.

2 Related Work

The Adaptive Weight Ranking Strategy (AWRP) algorithm proposed by Swain et al. [12] designs a weight model according to the frequency and access time of cached objects, and sorts them according to the weight values, giving priority to replacing the objects that are less frequently used and have not been visited for the longest time, but when the cache space is small, the algorithm works similarly to the LRU algorithm. Xuan et al. [13] proposed the Tachyon distributed file system, which is a component of the Spark ecosystem. It decouples the memory storage functionality from Spark and strives to achieve higher operational efficiency through a more granular division of labor. However, in its implementation, the LRU algorithm is still employed as the replacement algorithm. Chen et al. [14] proposed an elastic data persistence strategy to speed up data access by compressing data and reducing disk throughput, but data compression consumes CPU resources and affects the task execution rate. Xu et al. [15] proposed a dynamic memory manager, MEMTUNE, which dynamically adjusts the size of the cache space according to the workload and data caching requirements, but MEMTUNE uses Stage as the smallest unit in the replacement, and this coarse-grained mechanism is not suitable for nodes with small memory space. Zhang et al. [16] proposed a disk-based caching method that improves the way the disk handles data partitions and avoids excessive caching, but it does not fundamentally improve the efficiency of in-memory caching. Yu et al. [17] proposed an LRC algorithm to represent the importance of each data partition based on the number of times it is used, as a measure of the value of data partitions, but the number of citations alone cannot accurately measure the value of RDD. Khan et al. [18] designed a hierarchical hybrid memory system that utilizes Non-Volatile Memory (NVM) to cache RDD partitions, which requires modification at the hardware level and is not universal. References [19,20] proposed a weight model, which took the computational cost, number of uses, partition size, and life cycle of RDD as the influencing factors to measure RDD weight, and used the weight value for cache replacement, but did not propose a reasonable calculation method for the computational cost, and the weight model still needs to be improved. Drawing inspiration from [21,22], the WCSRP algorithm was developed by incorporating a weight factor for localization level during task execution. Nevertheless, the method used to calculate the cost of partitions lacks precision and is unable to accurately capture the weight of individual partitions. He et al. [23] proposed a WR algorithm to calculate the weight of RDD partitions by using the number of times and the size of partitions, considering the computational overhead of RDD partitions, but the considerations were not comprehensive. Sandholm et al. [24] proposed an Adaptive Cache Management (SACM) mechanism to identify and cache high-value RDDs by using frequency, RDD computational complexity, and capacity size to calculate weights, but it has limitations in using RDDs as the minimum granularity when buffer replacement [25].

To sum up, the current cache replacement algorithm for Spark is not perfect enough, and the mainstream Spark cache replacement algorithm evaluates the value of RDD partitions by establishing a weight model, and the current weight replacement algorithm still needs to be improved, such as the incomplete consideration of the influencing factors of RDD weight, and the unreasonable calculation method of weight value.

3 Spark Memory Replacement Strategy Based on Data Value

3.1 Spark Memory Optimization Strategy

The default cache replacement algorithm used by Spark itself is LRU, which is the least recently used algorithm. It comes from the traditional operating system environment and has good universality and performance. The LRU algorithm may be good in terms of hit rate, but in Spark, it is not only the hit rate. For example, as shown in Fig. 1 for a Spark data processing job, three intermediate result data need to be cached in memory: RDD_1, RDD_2, and RDD_3. However, the memory size only supports the caching of one of the data and the data storage difference of the data itself is not considered now. Assuming that the dimensions are the same, the access sequence of the three data in the calculation is RDD_1, RDD_2, RDD_2, RDD_2, RDD_1, RDD_3, and RDD_3. The calculation cost of RDD_1 is 10, the calculation cost of RDD_2 is 1, and the calculation cost of RDD_3 is 1. Now consider the hit and recovery cost of the LRU strategy and the strategy that uses the computational cost as the cache factor.

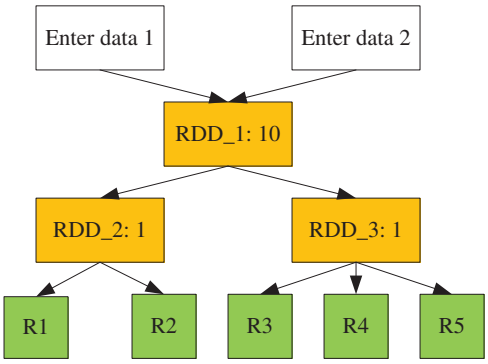

Figure 1: Schematic diagram of Spark data

For Spark's cache replacement strategy, we need to be more aware of the value of the data, set better cache replacement factors, and formulate a more efficient caching strategy. In Spark, considering the workflow of Spark and the characteristics of RDD caching, this paper puts forward several reference factors as the value of data:

(1) Calculate the cost *Cost_com*. Different RDD blocks have other calculation costs. We use the calculation generation time *Tcost* of the RDD block to represent the calculation cost.

(2) Recovery cost *Cost_re*. In Spark, if the RDD block is replaced from memory, there will be two results: Save it to disk or delete it directly. If the required RDD block is missed in the memory, it will search in the disk, and if it exists, the RDD will be restored to the memory, then the I/O time of disk and memory is the cost of recovery. If it does not exist, it needs to be recalculated to generate the RDD block, and the recovery cost is approximately equal to the calculation cost. Therefore, in general, the cost of recovery can be expressed as *Cost_re* the cost *Cost_com*.

(3) Storage cost *Cost_mem*. The storage cost compares the impact of the data itself on memory usage. The factors that affect the storage cost include the size of the RDD data Mem and the number of times the data has been used F. In theory, the larger the memory of the data, the less suitable it is to be replaced and the greater the storage value. So the storage value is proportional to the size of the data. The more used times, the fewer remaining visits, and the smaller the storage value. So the storage value is inversely proportional to the number of times the data is used. Therefore, the storage cost is expressed by *Mem*/*F*.

Based on the analysis of the partitioning weight factors, we can determine their impact on the RDD caching weight. The weight value of RDD can be calculated based on each weight factor. The calculation formula is shown in Eq. (1).

$$Value\left(p_{ij}\right) = \frac{\varphi \times US\left(p_{ij}\right) \times GC\left(p_{ij}\right)}{PS\left(p_{ij}\right) \times LC\left(p_{ij}\right)} \tag{1}$$

where φ is a calibration parameter, and US represents the usage frequency of RDD partitions.

In the actual use of Spark's default caching replacement algorithm, there is a problem that it is easy to replace high-value RDD partitions [26]. Based on the weight changes of RDD partitions during the task execution process, this chapter proposes a replacement strategy based on data value. The cached RDD partition information in memory is saved in a linked list, where the elements are the weight values of RDD partitions, sorted in ascending order. The partition information and weight values of all RDD partitions generated in this Spark job are stored in a hash table, including parameters such as generation cost, usage count, partition size and survival cycle, and dependency relationships [27–33].

The pseudo-code for the weight update algorithm is as follows:

Algorithm 1

```
Input:    p_wait: Partition to be cached;
Output: Blanklist: List of worthless partitions;
1: FatherSet=pMap[p_wait].FatherSet;
2: for i in FatherSet do
3:          if p_wait.operator()=Action then
4:                  i.num-;
5:                  i.lc-;
6:          else
7:                  i.num-;
8:          if if(i.num!=0) then
9:                updatelist.add(i);
10:       else
11:             BlankList.add(i);
12: for i in updateList do
13:          i.update();
14: return BlankList;
```

From the analysis of the LRU replacement strategy in Fig. 2, it can be seen that when a partition is used 0 times, it indicates that it is no longer in use, but may still occupy memory resources. Therefore, a cache-cleaning algorithm is proposed to promptly evict useless RDD partitions from memory to avoid wasting internal resources. The pseudo-code for the cache-cleaning algorithm is as follows:

Algorithm 2

```
Input:    Blanklist: List of worthless partitions; Memlist: Memory queue;
1: if BlankList.length==0 then
2:          return;
```

(Continued)

Algorithm 2 (continued)

```
new BlankList = new List;
for i in BlankList do
        if i.us=0 then
            Memlist.delete(i);
        FatherSet=pMap[i].FatherSet;
        for j in FathSet do
            if j.us==0 then
                newBlankList.add(j);
Recursively call the memory cleaning algorithm with newankst as input
```

The weight replacement algorithm is responsible for comparing the weight values of the existing partitions in memory with the weight values of the target partition when memory is low, and determining whether to replace the target partition [34–38]. The main purpose is to select partitions with smaller weight values, that is, to evict partitions with lower job values, to avoid a large amount of repeated calculation overhead caused by the eviction of partitions with high usage times or high generation costs. The pseudo-code for the weight replacement algorithm is as follows:

Algorithm 3

```
Input:  Memlist: Memory queue; p_wait: Partition to be cached; free: Remaining memory space;
Output: True/False
p_wait.update(); // Update weights based on Algorithm 1
if free>size(p_wait) then  // Sufficient memory remaining space
        free=free-size(p_wait);
        Memlist.add(p_wait);
        return true;
rpList=new List; //List to be replaced
rpListSize=0; //The total size of the list to be replaced
for i in Memlist do
        if weight(i)< weight(p_wait) then
            rpList.push(j);
            rpListSize= rpListSize+size(i);
            if rpListSize+free>=size(p_wait) then
                break;
        else
            break;
 if rpListSize+free<size(p_wait) then
        return false;
 else
        for i in relist do
            Memlist.delete(i);
        Memlist.push(p_wait);
        return true;
```

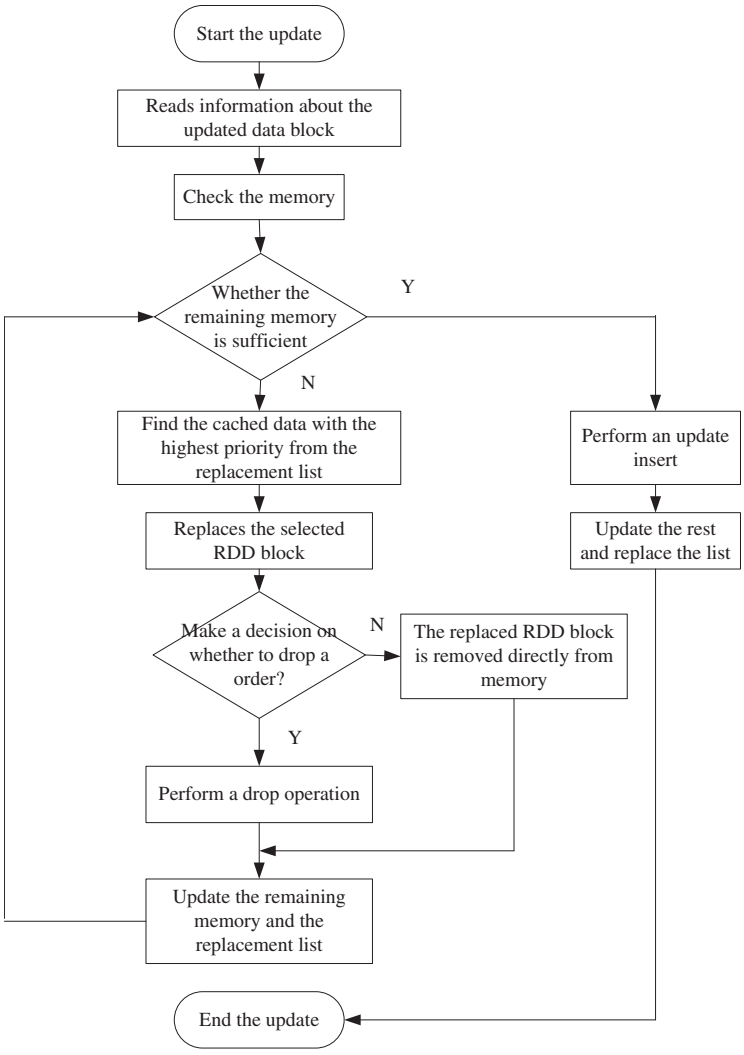

Figure 2: Cache replacement flow chart

The cache replacement strategy based on the value of the data is represented by Value in this article. The new cache replacement flowchart is shown in Fig. 2.

The rule of cache replacement is every time a new data block needs to be added to the memory, first, check whether the remaining memory space is sufficient for storage. If it is satisfied, directly add the memory to perform the update operation. At the same time, add the information of the newly inserted data block to the replacement list, and update the replacement list. Otherwise, it is necessary to replace the cache, find the data with the highest priority from the replacement list, obtain the corresponding RDD block ID, and eliminate the RDD data block from memory. Afterward, judge whether it needs to be placed on the disk according to the data value of the RDD block. If necessary, save the RDD block to the disk. If not, delete the data from memory directly and update the remaining memory. The value and replacement list are rechecked for memory until the update insertion is complete.

4 Evaluation

4.1 Experimental Configurations

The experiment uses three servers to simulate a distributed environment, with one server as the master node and the other two as the worker nodes. The configuration of each node is shown in Table 1.

Table 1: Hardware configuration

Server	1	2	3
Node	Master	Worker	Worker
IP address	192.168.1.2	192.168.1.3	192.168.1.4
Operating system	Centos 6.5	Centos 6.5	Centos 6.5
Memory	12 GB	12 GB	12 GB
Disk	600 GB	600 GB	600 GB
Central processing unit (CPU)	Intel® Xeon® X5560 @2.8 GHz	Intel® Xeon® X5560 @2.8 GHz	Intel® Xeon® X5560 @2.8 GHz
CPU core	8	8	8

The experiment is built with Hadoop + Spark, the Hadoop version is 2.8.1, and the Spark version is 2.1.1. The test software uses HiBench. HiBench, as a benchmark test framework for testing Hadoop and Spark, provides Hive: (aggregation, scan, join), and sorting (sort, TeraSort) [39]. Basic big data algorithms (WordCount, PageRank, nutchindex), Machine learning algorithms (K-Means, Bayes), cluster scheduling (sleep), throughput (dfsio), and streaming (Streaming) have a total of 19 workloads [40]. The application loads used in the experiments in this chapter are mainly the following three types: WordCount, PageRank, and K-Means.

4.2 The Experiment with Cache Replacement Strategy

For the convenience of description, the cache strategy proposed in this article is represented by VALUE, which is a cache replacement strategy based on the data value. To reflect the advantages of VALUE, we compared first in first out policy (FIFO), LRU, VALUE (Data value cache replacement policy), and VALUE+ (combined LRU and VALUE cache replacement policy) four cache replacement policy.

(1) Experimental Content

The experiment performs three different applications to test the performance of the optimized caching strategy. For each application, the input data size is used as the control variable, and the specific indicators include the sample size, the size of the input file of each sample, the maximum number of iterations, etc. Different applications involve different metrics. Compare the performance of different cache replacement strategies under the same input data volume size baseline.

In each application, to enrich the test results, this paper selects eight sets of test data for experimentation, experiments 5 times, and selects the most stable results as the performance results. The size of the amount of data in WordCount is only related to the size of the data, and the ten sets of data are shown in Table 2 below.

Table 2: Wordcount data scale variable table

The group number	1	2	3	4	5	6	7	8
Data size	32 K	320 M	1 G	3 G	4 G	5 G	6 G	8 G

The ten sets of data in PageRank are shown in Table 3.

Table 3: PageRank data size variable table

The group number	1	2	3	4	5	6	7	8
Number of pages	50	500	5000	50000	500000	500000	1000000	2000000
Number of iterations	1	2	3	3	3	10	10	10

K-Means is about the application of machine learning, which involves more data indicators. The size of ten sets of data is shown in Table 4.

Table 4: K-Means data size variable table

The group number	1	2	3	4	5	6	7	8
Number of clusters	5	5	5	5	5	5	5	5
Dimensions	3	3	3	10	10	20	20	20
Sample size	3000	30000	300000	300000	3000000	3000000	10000000	20000000
Enter the file size for each sample	600	6000	60000	60000	600000	600000	2000000	4000000
The maximum number of iterations	5	5	5	5	5	5	5	5
K	10	10	10	10	10	10	10	10
Convergedist	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5

In the cache strategy of LRU+Value, setting different substitution range thresholds has different results. Here is an experiment comparing the results at three thresholds:

1) The threshold is 5, indicated as LV5;

2) The threshold is 10, which is indicated as LV10;

3) The threshold is 20, which is expressed as LV20.

(2) Experimental Results

In the results, we use different colored dots and lines to represent other cache replacement policies and run time to represent running performance. The specific effects and expressions are as follows in Fig. 3.

1) WordCount application

As shown in the Fig. 3, when the amount of data is less than 1 G, the difference between the caching strategies is not very obvious, and LRU still shows good performance. When the amount of data gradually increases, the cache replacement strategy based on the value of the data begins to deliver better performance. Among the three different threshold caching strategies that combine the LRU and Value strategy, a point of 10 is better.

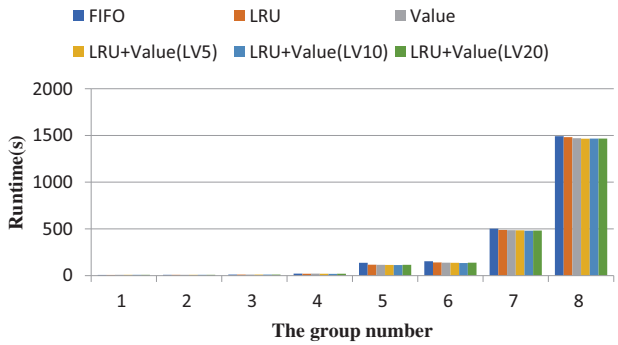

Figure 3: WordCount application performance

2) PageRank

Because the running time is enormous, to reflect the difference better, the data of groups 1–8 are represented by a line chart again as follows.

As shown in the Fig. 4, when the amount of data is small, the difference between the caching strategies is not very obvious, and LRU still indicates good performance. When the amount of data gradually increases, the number of data iterations increases, and the number of pages increases, the cache replacement strategy based on the value of the data begins to show better performance. Among the three different threshold caching strategies that combine LRU and Value strategy, the threshold value of 10 shows better performance.

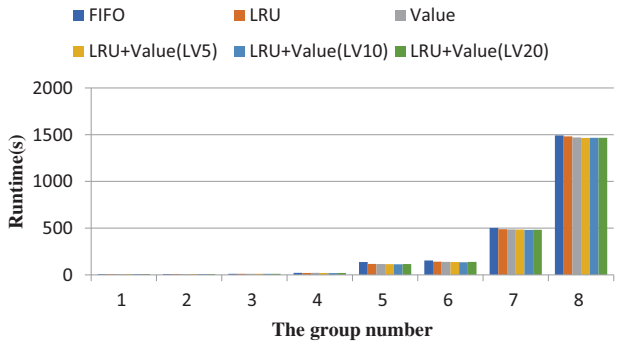

Figure 4: PageRank application performance (groups 1–8)

3) K-Means

As shown in the Fig. 5, it can be seen that when the amount of data is small, the difference between the cache strategies is not very obvious, and the LRU still performs well; As the data scale grows, cache substitution strategies based on the value of the data begin to show better performance, and in the cache strategies with three different thresholds combined with the LRU and Value policies, the threshold of 10 performs better.

Comparing the above three histograms, we find that when the experimental data is small, the best performance is LRU and FIFO instead of the strategy proposed in this article. Combining with the data scale at this time and analyzing, we find that the memory is sufficient. During the execution of the task, there is a no-cache replacement. Compared with the default strategy, the calculation of the

RDD data value table and the cost of the value replacement priority list are extra. Therefore, the performance of the algorithm designed for cache replacement proposed in this article is not the best.

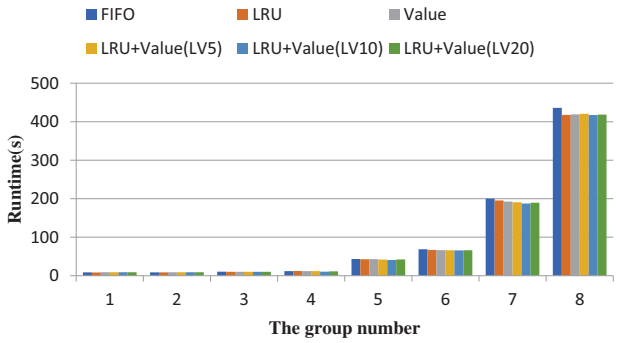

Figure 5: K-Means application performance (groups 1–8)

When the data scale reaches a certain level, the cache replacement strategy Value for the data value begins to show performance similar to LRU. After analysis, we found that there may be no significant data skew on the input data due to multiple experiments and stable results. Combined with the log, the difference between the storage value of RDD is not particularly large, so the cache replacement strategy for the value of data does not achieve excellent results. Moreover, the comparison found that the performance of LRU+Value is relatively better. On the one hand, LRU has a high hit rate and better applicability. On the other hand, combined with comparing data values, the performance of cache replacement and recovery is better.

In the three thresholds set, compared to the other two, the overall performance of LRU+Value (LV10) is better, the analysis is related to the task division of Spark itself, in the above experimental applications, according to logging, a stage is divided into 30 jobs to execute, so the threshold set to 10 effect is better. A threshold setting too low performance is closer to LRU because there is less range to choose from; If the threshold is set to 20 and greater, the processing process of the entire algorithm is closer to Value.

However, although the LRU+Value strategy is improved compared to the default strategy, the increase is not particularly large. In this regard, we conducted research and analysis mainly for two reasons. Conversely, the value strategy can exert better performance when the stage division is relatively unbalanced, and the value and advantages of cache replacement can be more reflected. The applications used in the experiment are pretty balanced. On the other hand, LRU needs to generate and maintain a list, and the Value policy also needs to calculate the cost of the list, although LRU+Value takes into account the advantages of these two strategies, the additional overhead is also increased, and the expected performance is also reduced.

To verify the effectiveness of the strategies proposed in this article, we compared the LRU algorithm, WCSRP algorithm [41], and LRU+Value cache replacement methods for the same test task. We selected the standard dataset provided by SNAP [42] for experimental data, and the dataset information is shown in Table 5.

Table 5: SNAP dataset information

Name	Nodes	Edges	Description
soc-sign-epinions	131828	841372	Epinions signed social network
Amazon0302	262111	1234877	Amazon product co-purchasing network from March 02 2003
Web-Stanford	281903	2312497	Web graph of Stanford.edu
Amazon0601	403394	3387388	Amazon product co-purchasing network from June 01 2003
com-youtube	1134890	2987624	YouTube online social network
Wiki-Talk	2394385	5021410	Wikipedia talk (communication) network

We used the PageRank task for the experimental task. The page ranking task is a typical data-intensive job that involves multiple iterations during its run process. As the RDD partitions are often replaced, it is easier to analyze the effectiveness of the proposed replacement algorithm. To avoid experimental results being accidental, we ran all experiments 10 times independently and took the average value as the final result. The experimental results are shown in Fig. 6.

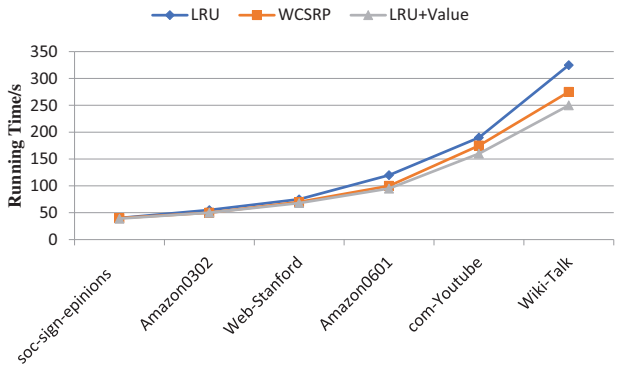

Figure 6: SNAP dataset execution time comparison

From the Fig. 6, it can be observed that as the number of nodes and edges in the dataset increases, there is a significant difference in the task execution time among the various cache replacement algorithms. When running on the soc-sign-epinions dataset, due to the small amount of computation, the three algorithms show little difference in execution time. However, as the computation workload increases, both LRU+Value and WCSRP show a noticeable increase in execution time. The average running time of the LRU+Value algorithm is 18.17% higher than the default LRU algorithm and 7.61% higher than the WCSRP algorithm. This is because when the data volume is small, fewer RDD partitions need to be cached in the application, and the internal memory space of the task executor is sufficient. During the execution process, there are rarely occurs cache replacements, so the differences between the cache replacement algorithms are barely noticeable, resulting in a small difference in task completion time. As the dataset increases, the workload increases and cache replacements continue to occur, the LRU+Value algorithm can better preserve high-value RDD partitions, shortening the task completion time.

To further verify the effectiveness of the LRU+Value algorithm, multiple datasets from Tables 4, 5 were combined into a single dataset to validate the impact of the algorithm on Spark platform performance. The iteration times of the task were gradually increased from 2 to 10 times. To avoid accidental results, each iteration was executed 10 times, and the average execution time was taken as the final result, as shown in Fig. 7.

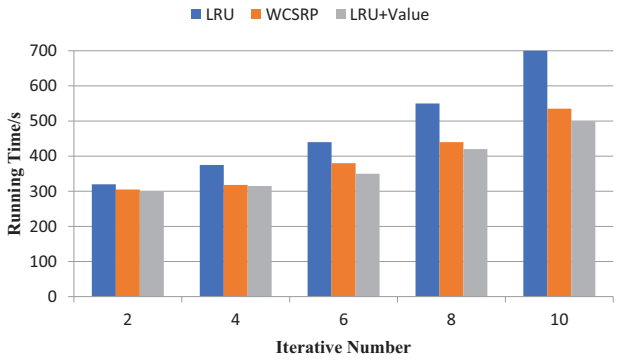

Figure 7: Comparison of iterative calculation execution time

Fig. 7 compares the execution time of iterative calculations among three algorithms. It can be observed that as the iteration count increases, the difference in execution time among the three algorithms increases. The LRU algorithm only uses the parameter of recent usage frequency to evaluate the value of a partition, which is not able to accurately assess the value of the partition for the entire job. This results in high-value partitions being evicted and additional time being incurred. As the task workload increases, the execution time increases rapidly, which is also reflected in the figure. In contrast, the execution time of the WCSRP and LRU+Value algorithms increases relatively steadily. Although WCSRP has a significant improvement compared to LRU, its calculation of weight factors such as the computational cost is too simple, which affects the accuracy of weight values. The DWCR algorithm considers more comprehensive weight factors for RDD partitions compared to WCSRP and updates the partition weights as the application executes, allowing the most valuable RDD partitions to continue to be cached in memory, efficiently utilizing memory space. Therefore, under multiple task sets, the execution time is 5.41% lower than that of the WCSRP algorithm and 18.33% lower than that of the LRU algorithm. As a result, the algorithm proposed in this article considers weight factors for RDD partitions more comprehensively and accurately, reducing job execution time and significantly improving job execution efficiency.

Through monitoring the iterative experiments, the cache hit rates of the three algorithms are calculated and presented in Table 6.

Table 6: Cache hit rate comparison

	LRU	WCSRP	LRU+Value
Cache hit rate	63.8%	72.6%	75.1%

As can be seen from the table, when the task workload increases, the LRU algorithm replaces a larger number of partitions, resulting in a low cache hit rate that affects the job execution efficiency.

Compared to the existing improved algorithm WCSRP, the LRU+Value algorithm has a slight increase in hit rate, which is consistent with the task execution time shown in Fig. 7.

In the Spark's Storage module, which is responsible for storage management, new classes LoadPredict and BlockInfo have been introduced to encapsulate load prediction and RDD block information respectively. Additionally, modifications and enhancements were made to methods within the BlockManager and BlockEvictionHandler to integrate the cache replacement mechanism proposed in this paper into a real-world Spark cluster.

A Spark Executor with 2 GB of memory was configured within the cluster, and the task completion time during the execution of iterative applications was tested, comparing it against the default cache replacement mechanism in Spark, which is the LRU algorithm. To ensure fairness in the experiments, each set of experiments was repeated five times, with the average value taken as the final result. The experiments employed five distinct datasets: Com-amazon, Amazon0302, Amazon0601, com-youtube, and Wikipedia talk network, sourced from literature [40–42]. The computational complexity and completion time of the dataset are related to the number of nodes and edges, and the corresponding relationship between the number of nodes and edges is shown in Table 7.

Table 7: Information on the number of nodes and edges in the dataset

Dataset	Number of nodes	Number of edges
com-amazon	334863	925872
Amazon0302	410236	3356824
Amazon0601	403394	3387388
com-YouTube	1134890	2987624
Wikipedia talk network	2394385	5021410

The same iterative application was run on all five experimental datasets, ensuring an equal number of iterations across each experiment. The experimental results are illustrated in Fig. 8.

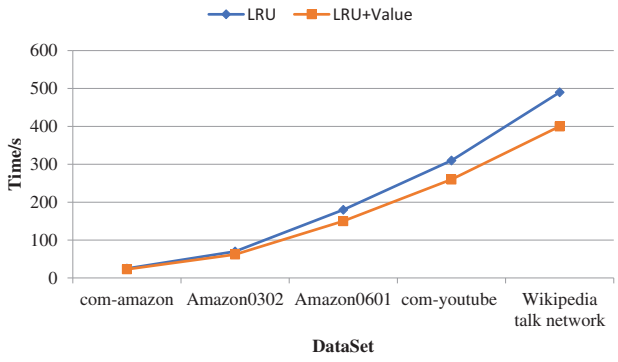

Figure 8: Comparison of task completion times on different datasets in a real cluster

The experimental results indicate that there are noticeable differences in task completion times across different datasets. When executed on datasets with larger numbers of nodes and edges, resulting in greater computational loads, both the LRU algorithm and the mechanism proposed in this paper exhibited longer completion times compared to when executed on smaller datasets. However,

regardless of whether running on large or small datasets, the mechanism proposed in this paper consistently demonstrated a slight improvement in task completion times over the LRU algorithm. Moreover, as the scale of the dataset increased, the advantage of the proposed mechanism became more pronounced. This is because when dealing with smaller datasets where the data file sizes are also relatively small, the storage space within the Executor's memory is ample, leading to almost no cache replacement occurring during iterations. Consequently, the benefits of the proposed mechanism could not be fully leveraged, resulting in task completion times similar to those of the LRU algorithm. Conversely, when handling larger datasets where cache replacements occur frequently, the proposed mechanism can better utilize memory space, thereby significantly reducing the task completion time.

5 Conclusion

With the advent of the significant data era, more and more data information needs to be processed, bringing substantial challenges to storage and computing. The amount of data is getting larger and larger, and the I/O bottleneck of computing and scheduling from the disk has increasingly become an essential factor restricting performance. The Spark came into being and proposed in-memory computing, significantly improving the computing speed. It is a distributed computing framework based on memory. The effective use of memory is one of the critical factors in the performance of Spark. Among them, Spark's cache replacement mechanism is an essential aspect of Spark system management and the use of memory resources. This paper studies the Spark memory optimization method based on NVM, optimizes the original cache replacement strategy, defines the data value of the cached RDD, and proposes a Value strategy that is more in line with the characteristics of Spark itself, combined with the original LRU, and proposes The LRU+Value is calculated. A threshold with better results is obtained through experiments. At the same time, the broadcast block and memory parameters generated by Spark's tasks are optimized.

The paper describes the improved scheduling algorithm and cache replacement algorithm for the Spark platform, which have led to enhanced system execution efficiency. However, further optimization of the cache replacement algorithm has only been validated in a homogeneous cluster. In the next step, it will be deployed to a heterogeneous cluster to verify the effectiveness of the dynamic weight-based cache replacement algorithm. By exploring these adaptations, the process would involve extensive research, design, development, and validation stages to ensure successful integration and optimal performance within these new contexts. We aim to demonstrate the versatility and broader impact potential of our methods, highlighting their value in enhancing the performance and efficiency of modern distributed computing ecosystems beyond the original scope of Spark.

Acknowledgement: The authors would like to thank the editors and reviewers for their valuable work, as well as the supervisor and family for their valuable support during the research process.

Funding Statement: This work is supported by the National Natural Science Foundation of China (61872284); Key Research and Development Program of Shaanxi (2023-YBGY-203,2023-YBGY-021); Industrialization Project of Shaanxi Provincial Department of Education (21JC017); "Thirteenth Five-Year" National Key R&D Program Project (Project Number: 2019YFD1100901); Natural Science Foundation of Shannxi Province, China (2021JLM-16, 2023-JC-YB-825); Key R&D Plan of Xianyang City (L2023-ZDYF-QYCX-021)

Author Contributions: The authors confirm contribution to the paper as follows: Qinlu He: Methodology, Investigation, Software, Writing, Funding. Fan Zhang: Investigation, Writing-Original Draft,

Writing-Review and Editing. Genqing Bian, Weiqi Zhang and Zhen Li: Resources, Validation, Writing-Review and Editing. All authors reviewed the results and approved the final version of the manuscript.

Availability of Data and Materials: This article uses the experimental data set is open-source. Data source address: https://www.github.com/.

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

References

[1] D. Reinsel, J. Gantz, and J. Rydning, "The digitization of the world from edge to core," in *IDC White Paper*, 2018.

[2] M. AlJame, I. Ahmad, and M. Alfailakawi, "Apache Spark implementation of whale optimization algorithm," *Cluster Comput.*, vol. 23, pp. 2021–2034, 2020.

[3] M. Zaharia, *An Architecture for Fast and General Data Processing on Large Clusters*. Association for Computing, Machinery and Morgan & Claypool, 2016.

[4] M. X. Duan, K. Li, Z. Tang, G. Q. Xiao, and K. Q. Li, "Selection and replacement algorithms for memory performance improvement in Spark," *Concurr. Comput. Pract. Exp.*, vol. 28, no. 8, pp. 2473–2486, 2016. doi: 10.1002/cpe.3584.

[5] Y. Geng, X. H. Shi, C. Pei, H. Jin, and W. B. Jiang, "LCS: An efficient data eviction strategy for Spark," *Int. J. Comput. Inf. Sci.*, vol. 45, no. 6, pp. 1285–1297, 2017. doi: 10.1007/s10766-016-0470-1.

[6] Z. Y. Yang, D. L. Jia, S. Ioannidis, N. F. Mi, and B. Sheng, "Intermediate data caching optimization for multi-stage and parallel big data frameworks," *Future Gener. Comput. Syst.*, vol. 27, no. 5, pp. 543–549, 2018. doi: 10.1109/CLOUD.2018.00042.

[7] P. Bailis, C. Fournier, J. Arulraj, and A. Pavlo, "Research for practice: Distributed consensus and implications of NVM on database management systems," *Commun. ACM*, vol. 59, no. 11, pp. 52–55, 2016. doi: 10.1145/2949033.

[8] Q. L. He, G. Bian, W. Q. Zhang, F. L. Wu, and Z. Li, "TCFTL: Improved real-time flash memory two cache flash translation layer algorithm," *J. Nanoelectron. Optoelectron.*, vol. 16, no. 3, pp. 403–413, 2021. doi: 10.1166/jno.2021.2970.

[9] D. Nellans, M. Zappe, J. Axboe, and D. F. Fusionio, "Ptrim () + exists (): Exposing new FTL primitives to applications," in *2nd Annu. Non-Volatile Mem. Workshop (NVMW)*, La Jolla, CA, USA, UCSD, 2011.

[10] Y. Y. Lu, J. W. Shu, and L. Sun, "Blurred persistence: Efficient transactions in persistent memory," *ACM Trans. Storage*, vol. 12, no. 1, pp. 1–29, 2016. doi: 10.1145/2851504.

[11] X. H. Shi *et al.*, "Deca: A garbage collection optimizer for in-memory data processing," *ACM Trans. Comput. Syst.*, vol. 36, no. 1, pp. 1–47, 2018.

[12] D. Swain, B. Paikaray, and D. Swain, "AWRP: Adaptive weight ranking policy for improving cache performance," *Comput. Sci.*, vol. 35, no. 8, pp. 285–297, 2011.

[13] P. Xuan, F. Luo, R. Ge, and P. K. Srimani, "DynIMS: A dynamic memory controller for in-memory storage on HPC systems," 2016. doi: 10.48550/arXiv.1609.09294.

[14] D. Chen, H. P. Chen, Z. P. Jiang, and Y. Zhao, "An adaptive memory tuning strategy with high performance for Spark," *Int. J. Big Data Intell.*, vol. 4, no. 4, pp. 276–286, 2017. doi: 10.1504/IJBDI.2017.086970.

[15] L. N. Xu, M. Li, L. Zhang, A. R. Butt, Y. D. Wang and Z. Z. Hu, "MEMTUNE: Dynamic memory management for in-memory data analytic platforms," in *2016 IEEE Int. Parallel Distr. Process. Symp. (IPDPS)*, IEEE, 2016, pp. 383–392.

[16] K. H. Zhang, Y. Tanimura, H. Nakada, and H. Ogawa, "Understanding and improving disk-based intermediate data caching in Spark," in *2017 IEEE Int. Conf. Big Data (Big Data)*, Boston, MA, USA, 2017, pp. 2508–2517. doi: 10.1109/BigData.2017.8258209.

[17] Y. H. Yu, W. Wang, J. Zhang, and K. B. Letaief, "LRC: Dependency-aware cache management for data analytics clusters," in *IEEE INFOCOM 2017-IEEE Conf. on Comput. Commun.*, Atlanta, GA, USA, IEEE, 2017, pp. 1–9.
[18] M. M. Khan, M. A. U. Alam, A. K. Nath, and W. K. Yu, "Exploration of memory hybridization for RDD caching in Spark," in *Proc. 2019 ACM SIGPLAN Int. Symp. Mem. Manag.*, Phoenix, AZ, USA, ACM, 2019, pp. 41–52.
[19] C. L. Li, Y. Zhang, and Y. L. Luo, "Intermediate data placement and cache replacement strategy under Spark platform," *J. Parallel Distr. Comput.*, vol. 2022, no. 13, pp. 114–135, 2022. doi: 10.1016/j.jpdc.2022.01.020.
[20] Q. He, B. Shao, and W. Zhang, "Data deduplication technology for cloud storage," *Tehnički vjesnik*, vol. 27, no. 5, pp. 1444–1451. doi: 10.17559/TV-20200520034015.
[21] R. Myung and S. Choi, "Machine-learning based memory prediction model for data parallel workloads in apache Spark," *Symmetry*, vol. 13, no. 4, pp. 697–713, 2021. doi: 10.3390/sym13040697.
[22] B. T. Rao and L. S. S. Reddy, "Scheduling data intensive workloads through virtualization on MapReduce based clouds," *Int. J. Comput. Sci. Net.*, vol. 13, no. 6, pp. 105–112, 2013.
[23] Q. L. He, G. Q. Bian, W. Q. Zhang, F. Zhang, S. Q. Duan and F. L. Wu, "Research on routing strategy in cluster deduplication system," *IEEE Access*, vol. 9, pp. 135485–135495, 2021. doi: 10.1109/ACCESS.2021.3116270.
[24] T. Sandholm and K. Lai, "Dynamic proportional share scheduling in hadoop," in *Int. Conf. Job Sched. Strat. Parallel Process.*, Atlanta, GA, USA, 2010, pp. 110–131.
[25] Q. He *et al.*, "File block multi-replica management technology in cloud storage," *Cluster Comput*, vol. 27, pp. 457–476, 2024. doi: 10.1007/s10586-022-03952-1.
[26] Q. L. He, P. Z. Gao, F. Zhang, G. Q. Bian, W. Q. Zhang and Z. Li, "Design and optimization of a distributed file system based on RDMA," *Appl. Sci.*, vol. 13, no. 15, pp. 8670, 2023. doi: 10.3390/app13158670.
[27] A. Verma, L. Cherkasova, V. S. Kumar, and R. H. Cambell, "Deadline-based workload management for MapReduce environments: Pieces of the performance puzzle," in *Netw. Oper. Manag. Symp.*, Maui, HI, USA, 2012, pp. 900–905. doi: 10.1109/NOMS.2012.6212006.
[28] H. J. Li, H. C. Wang, A. P. Xiong, J. Lai, and W. H. Tian, "Comparative analysis of energy-efficient scheduling algorithms for big data applications," *IEEE Access*, vol. 6, pp. 40073–40084, 2018. doi: 10.1109/ACCESS.2018.2855720.
[29] N. Zacheilas and V. Kalogeraki, "Real-time scheduling of skewed MapReduce jobs in heterogeneous environments," in *ICAC 2014*, Philadelphia, PA, USA, 2014, pp. 189–200.
[30] I. Gog *et al.*, "Broom: Sweeping out garbage collection from big data systems," in *Usenix Conf. Hot Topics Oper. Syst.*, Switzerland, 2015.
[31] L. Yan *et al.*, "CoPIM: A concurrency-aware PIM workload offloading architecture for graph applications," in *IEEE/ACM Int. Symp. Low Power Electr. Des.*, IEEE, 2021. doi: 10.1109/ISLPED52811.2021.9502483.
[32] J. Liu, N. Ravi, S. Chakradhar, and M. Kandemir, "Panacea: Towards holistic optimization of MapReduce applications," in *CGO '12 Proc. Tenth Int. Symp. Code Gen. Opt.*, San Jose, California, USA, 2012, pp. 33–43.
[33] Q. A. Chen, F. Li, Y. Cao, and M. S. Long, "Parameter optimization for Spark jobs based on runtime data analysis," (In Chinese), *Comput. Eng. & Sci.*, vol. 2016, pp. 11–19, 2016.
[34] J. G. Xu, G. L. Wang, S. Y. Liu, and R. F. Liu, "A novel performance evaluation and optimization model for big data system," in *Int. Symp. Parallel Distrib. Comput.*, Fuzhou, China, 2016, pp. 121–130.
[35] Y. Y. Lu, J. W. Shu, J. Guo, S. Li, and O. Mutlu, "LightTx: A lightweight transactional design in flash-based SSDs to support flexible transactions," in *Proc. IEEE 31st Int. Conf. Comput. Design (ICCD)*, Asheville, North Carolina, USA, IEEE, 2013, pp. 115–122.
[36] Q. L. He, G. Q. Bian, B. L. Shao, and W. Q. Zhang, "Research on multifeature data routing strategy in deduplication," *Sci. Program.*, vol. 2020, no. 10, pp. 118–132, 2020. doi: 10.1155/2020/8869237.

[37] I. O. Fusion, "The fusion-io difference," Accessed: May 6, 2015. [Online]. Available: http://www.fusionio.com/load/-media-/lqaz4e/docsLibrary/FIO_SSD_Differentiator_Overview.pdf.
[38] J. Yang, D. B. Minturn, and F. Hady, "When poll is better than interrupt," in *Conf. File Storage Tech. (FAST)*, San Jose, CA, USA, USENIX, 2012, pp. 25–32.
[39] Z. H. Cheng, W. Shen, W. Fang, and C. W. Lin, "A parallel high-utility item set mining algorithm based on hadoop," *Complex Syst. Model. Simul.*, vol. 3, no. 1, pp. 47–58, 2023. doi: 10.23919/CSMS.2022.0023.
[40] S. Kumar and K. K. Mohbey, "UBDM: Utility-based potential pattern mining over uncertain data using Spark framework," in *Int. Conf. Emerg. Tech. Comput. Eng.*, Cham, Springer, 2022.
[41] M. X. Duan, K. L. Li, Z. Tang, G. Q. Xiao, and K. Q. Li, "Selection and replacement algorithms for memory performance improvement in Spark," *Concurr. Comput. Pract. Exp.*, vol. 28, no. 8, pp. 2473–2486, 2016. doi: 10.1002/cpe.3584.
[42] J. Leskovec, "Stanford network analysis project," Accessed: Dec. 23, 2018. [Online]. Available: http://snap.stanford.edu/data/index.html.