https://doi.org/10.31871/IJNTR.5.9.10
International Journal of New Technology and Research (IJNTR)
ISSN: 2454-4116, Volume-5, Issue-9, September 2019 Pages 27-32

Research Cache Replacement Strategy in Memory Optimization of Spark

Caili Zhao, Yong Liu, Ximei Du, Xuezhen Zhu

***Abstract*—In Spark, using LRU to implement RDD cache replacement. Its metrics do not take the data characteristics of Spark into account, resulting in memory not being effectively utilized, affecting task execution efficiency. This paper optimizes the LWR (Least Weight Replacement) algorithm, and a new replacement algorithm is proposed. Considering the parallel computing, the dependency integrity impact factor is added to the weight calculation to make the RDD partition weight value more accurate, so as to improve the accuracy of the cache replacement object selection, and the relevant factor values are dynamically adjusted according to the task execution, so that the cache replacement can adapt to the changes in the task execution process. The source of the experimental data set for this article is the Stanford Network Analysis Project. According to comparison experiments, this methods can effectively improve task execution efficiency.**

***Index Terms*—Parallel computing, Resilient Distributed Dataset, Spark, self-adaptive**

I. INTRODUCTION

In the big data environment, data is exploding, data types are complex and diverse, and the processing of data requires powerful technical means. Spark [1] parallel computing engine quickly occupied the market with its four advantages: high efficiency [2], ease of use, versatility and compatibility. Spark speeds up batch processing tasks through sophisticated memory calculation and processing mechanisms. Spark speeds up batch processing tasks through sophisticated memory calculation and processing mechanisms. On the one hand, Spark makes full use of the cluster's memory: using Resilient Distributed Datasets (RDD) as the data structure [3], whose all partitions can be processed in parallel, using distributed memory to cache RDD intermediate results, which makes Spark have a bigger advantage than other parallel frameworks in handling iterative machine learning. On the other hand, Spark uses a Directed Acyclic Graph (DAG) to record the dependencies between RDDs, which can quickly recover the lost RDD partitions, as illustrated in Fig.1.1.

The Spark memory caching mechanism uses LRU(Least Recently Used) to replace the block. The metrics method in LRU only considers the time factor, and does not consider the RDD partition feature, which may result in the removal of the RDD partition with high reusability or high recalculation cost, which increases the task calculation time.

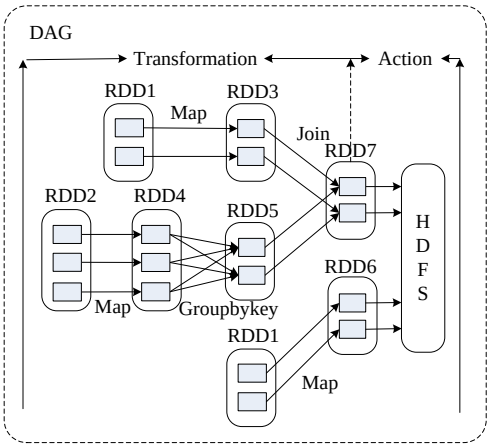


Fig.1.1 Directed acyclic graph of the Spark program

Recognizing this problem, many scholars have studied RDD cache optimization [4]. Duan et al. [4]proposed a selection and replacement (WR) algorithm to improve memory performance. Bian et al. [5] proposed a weight buffer replacement (LWR) algorithm. However, they ignore the factors that change the weight during execution. Jiang et al.[6] further consider whether durability is required by judging the cost and cost of calculation. On the basis of studying FIFO, LRU and other cache algorithms, Yu et al. proposed minimum reference count (LRC) [7] and minimum effective reference technique (LERC) [8].Ho et al.[9] proposed a cache update technique that enables users to replace a single RDD partition by partially updating RDD, thus avoiding the large overhead caused by loading the entire RDD. Swain et al. [10] designed an AWRP algorithm, calculating the weight according to the access frequency of the object. Chen et al. [11] proposed a register allocation (RA) replacement algorithm to replace the RDD partition with the latest end of use time. AWRP and RA algorithms do not consider such characteristics as RDD partition size and recalculation cost. Meng et al. [12] fully considered the distributed storage characteristics of RDD partitions and pointed out the impact of complete and incomplete RDD partitions on cache, but there was no parameter correction in weight calculation. Liu et al. [13]proposed a new RDD partition weight cache replacement algorithm, which comprehensively considers the major factors affecting RDD cache, improves the cache strategy, and improves the

Caili Zhao, School of Information Engineering, Henan University of Science and Technology, Louyang 471023, Henan, P. R. China
Yong Liu, School of Information Engineering, Henan University of Science and Technology, Louyang 471023, Henan, P. R. China
Ximei Du, School of Information Engineering, Henan University of Science and Technology, Louyang 471023, Henan, P. R. China
Xuezhen Zhu, School of Information Engineering, Henan University of Science and Technology, Louyang 471023, Henan, P. R. China.

execution efficiency of Spark. But it does not consider the integrity of partition dependencies.

In the above cache replacement strategy optimization research, although the computational performance is improved relative to the original Spark, the consideration of the RDD partitioning characteristics is still not comprehensive enough, so we improve the weight replacement algorithm. This method reduces the impact of memory resource bottlenecks and improves task execution efficiency. Compared with existing research, this method can further improve the performance of Spark computing.

The remainder of this paper is organized as follows. Section 2 introduces the background. Section 3 gives the cache replacement model of Spark and shows the proposed algorithm. The theoretical analysis and experiment are illustrated in Section 4. Finally, we make a conclusion in Section 5.

II. BACKGROUND

A. *Resilient Distributed Datasets*

Resilient Distributed Datasets (RDD) is the most basic abstraction of Spark. It is an abstract use of distributed memory and implements an abstract implementation of operating distributed datasets by manipulating local collections. Spark programming is built around the creation and execution of operations on the RDD. RDD is the core of Spark. An RDD is a collection of distributed objects. It is essentially a read-only collection of partition records. Each RDD can be divided into multiple partitions.

Fig.2.1 shows the program flow in Spark and the distributed storage of RDDs in the cluster. RDD supports four operations: creation, transformation, control, and action operations. The conversion operation builds most of the dependencies between the RDDs. When the RDD is partially lost for some reason, the missing RDDs can be recalculated based on the dependencies. In the conversion process, only the action operation takes place, the actual operation will be carried out. Control operations can cache some re-used RDDs into storage memory, effectively reducing computational costs.

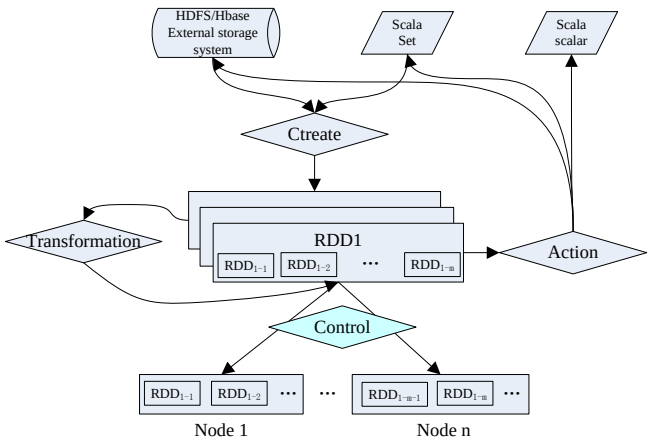


Fig.2.1 Program flow of RDD in Spark

B. *LRU Cache Replacement Strategy for RDD*

Currently, the Spark cache mechanism uses LRU replacement Strategy to replace block. LRU is a classic replacement algorithm, based on the assumption that data which not used for a long time is not likely to be used in the future. The operations of LRU has three types, such as insertion, search and replacement as showed in Fig.2.2. First, the newly added data is inserted into the head of the linked list. Second, the accessed data is moved to the linked list header. Third, when the storage space of the linked list is insufficient, the data at the end of the linked list is discarded.

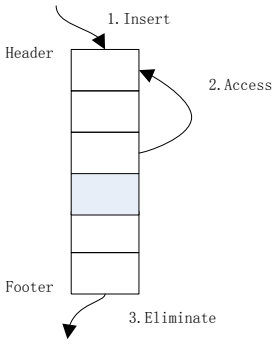


Fig.2.2 The principle of LRU

In Spark, the LRU replacement algorithm is implemented by a double-linked list feature. In Spark, different RDD partitions in the same storage memory are heterogeneous, that is, they are different in size and frequency of use. In this case, considering only the time factor leads to a lot of unnecessary calculations. Both cache replacement and memory recovery do not meet the requirements of task calculation. Therefore, it is necessary to develop a replacement algorithm based on the characteristics of RDD partitions, and dynamically adjust the value of the partition features to increase its adaptability and improve the accuracy of the replacement.

III. CACHE REPLACEMENT MODEL OF SPARK

Spark divides the RDD into multiple partitions and submits them to the worker nodes for parallel computing. In Spark, each task contains multiple RDDs, and the number of uses of each RDD may be different. Here, the set of <RDD, NR> of key-value pairs represents RDDs,

$$R = \{\langle RDD_1, NR_1\rangle, \langle RDD_2, NR_2\rangle, ..., \langle RDD_n, NR_n\rangle\} \quad (1)$$

In the formula, NR is the number of RDD used times in the task's executing process. Since each RDD consists of multiple partitions.

And we use R_{ij} to express the partition of RDD.

A. *Feature analysis*

(1) Frequency of use

Different RDDs in Spark have different access frequencies, and the frequency of access is reduced during the running of the program. When an action occurs, the DAG Scheduler creates a DAG based on the RDD's lineage. By traversing the DAG map, we use G<R, NR> to represent the

DAG, where NR represents the degree of RDD in the DAG graph.

The total frequency of the RDD (UF) which used during the entire task calculation process can reflect the importance of RDD to a certain extent, and the RDD with high total frequency of use should be preferred for caching. The specific formula is expressed as followed:

$$UF_{Ri} = N_{Ri} \quad (2)$$

Use a counter to record the number of times RDD has been used by monitoring the use of RDD, notes for VN , The RDD remaining usage frequency during the running of the program can be expressed as:

$$RVN_{Ri} = UF_{Ri} - VN_{Ri} \quad (3)$$

Initialization, the remaining usage of the RDD partition is equal to the remaining frequency of the RDD in which it resides. The specific formula is as follows:

$$RVN_{Rij} = RVN_{Ri} \quad (4)$$

(2) Partition size

When other attributes are consistent, the partitions occupying a large memory space should be preferentially deleted to release more resources. We use S_{Rij} to express the size of RDD partition.

(3) Calculation cost

When the cache memory is insufficient, the LRU release the least recently used partition by considering the time characteristics of the partition. When the deleted partition is reused, it needs to be recalculated, which will generate unnecessary computational overhead. Therefore, partitions with higher cost calculations should not be replaced. Here, we use the definition of the computational cost of the RDD partition in [5]. The partition is the parent RDD partition on which it depends and is generated by various operator calculations. $R_{ij}P$ is the parent RDD partition set of R_{ij} . The calculation time of the RDD partition is the sum of the read time and the execution time, namely:

$$T_{Rij} = RT_{RijP} + ET_{Rij} \quad (5)$$

If all the partitions in the $R_{ij}P$ collection are stored in memory, the data read time can be ignored， $RT_{RijP} = 0$.

Use the partition calculation time as the only indicator to measure the cost of the calculation, namely:

$$\mathrm{Cost}_{Rij} = T_{Rij} \quad (6)$$

(4) Dependency integrity

Data parallel tasks typically rely on multiple input data blocks. Unless all of these blocks are cached in memory, they won't speed up. PACMan [14] attempts to satisfy the "all or nothing" attribute of cache management in a parallel cluster. However, PACMan does not know the semantics of the job DAG. Its goal is to accelerate data sharing between different jobs by caching the complete data set (HDFS file).

The DAG diagram shown in Fig.1.1, RDD3 and RDD5 generate RDD7 through the Join operator. An RDD partition corresponds to one data block. Then the calculation of the partition RDD7-1 depends on RDD3-1 and RDD5-1, and the calculation of the partition RDD7-2 depends on RDD3-2 and RDD5-2. We call this a dependency group, and the dependency information is obtained from the data-dependent DAG. The generation of RDD7-1 is not accelerated unless the elements in the dependency group (RDD3-1, RDD5-1) are cached in memory. The elements in the dependency group are called peer partitions. If all the elements in the dependency group are cached in memory, they are called complete dependency groups. When other attributes are consistent, the partitions in the incomplete dependency group should be preferentially deleted.

Definition1 dependency integrity. The RDD dependency integrity is quantified to represent the dependent integrity of the RDD partition and is expressed as RC_{Rij} . If there is a dependency group containing R_{ij} , it is used to record the number of complete dependency groups, initialized to $RC_{Rij} = 0$. If there is no dependency group including R_{ij} , $RC_{Rij} = 1$ .then the value is unchanged during the running of the program.

B. Weight model

Considering the above-mentioned partition use frequency, partition size, calculation cost and partition dependency integrity, the weight model is constructed by linear weighted accumulation method. The weight calculation formula of the partition is as shown in formula (7):

$$W_{Rij} = aRVN_{Rij} + bS_{Rij} + c\,\mathrm{Cost}_{Rij} + dRC_{Rij} \quad (a,b,c,d > 0, a+b+c+d = 1) \quad (7)$$

W_{Rij} is the weight value of the No. j partition. a,b,c,d are used to adjust the weight of the above four decision factors, and the selection of the weight value is determined by the specific task requirements of the user.

C. Performance Evaluation Model

Definition2 Task execution speedup. Using S_{opt} to indicate the task execution speedup, which is used to measure the performance of Spark after optimization. The higher the speedup ratio, the better the Spark performance. The formula is as follows:

$$S_{opt} = \frac{T_{originalSpark}}{T_{opt}} \quad (8)$$

$T_{originalSpark}$ is expressed as the executing time of the task on the original Spark, T_{opt} is expressed as the executing time of the task on the Spark after optimistic.

D. Self-adaptive Weight Cache Replacement Algorithm

Aiming at the shortcomings of using LRU algorithm in Spark, the existing weight substitution algorithm LWR[6] is improved, and a new RDD cache replacement algorithm-Self-adaptive Weight Cache Replace Algorithm (SWCR) is proposed. The main idea of the algorithm is to compare the weight values to form a partition list to be replaced.

The pseudo code of the improved weight cache replacement algorithm is shown in Table 2. In this algorithm, after replacing the RDD partition, the weight of the RDD partition stored in the memory is updated for use in the next replacement.

Table 1 Self-adaptive Weight Cache Replacement Algorithm

Algorithm 1：Self-adaptive Weight Cache Replacement Algorithm SWCR

Input ： Weight map of RDD partition: *WList*
Partition is ready to cache:P
The Weight of P:W
The size of P: S

Initialized： The Replace list : rpList = new list
The sum sizes of the List ： rpListSize = 0

```
1. for((RP, WRP) < −WList)
2.     if (W > WRP) then
3.         rpList..add(RDD)
4.     end if
5. end for
6. if (rpList.lenght = 0) then
7.     return
8. end if
9. rpList.drderByWeight()
10.    for(i = 0 until rpList.length)
11.        rpListSize + = rpList[i].size
12.        if (rpListSize + RFMemSize > S) then
13.            for( j = 0 to i)
14.                dropBlock(rpList[ j])
15.                WList.delete(rpList[ j])
16.            end for
17.            cachePartition(p)
18.            WList.add(p, W)
19.            Re new WList
20.            return
21.        end if
22.    end for
23. rpList.clean()
```

The specific steps of Algorithm 1:

(1) Get the weight and size of the RDD partition to be cached.

(2) Perform conditional filtering on the cached RDD partition, and put the object whose weight value is smaller than the partition to be cached into the list to be replaced.

(3) If the list of partitions to be replaced is empty, the partition is not cached.

(4) Otherwise, the list of replaced partitions is sorted by weight. Traversing the replacement list, when the sum of the size of the free memory and the replacement list is larger than the size of the partition to be cached, the traversal is stopped, the traversed partition to be replaced is moved out of the cache, the memory is released, and the corresponding weight is also moved out of the weight mapping, after which Cache the partition to be cached and add the partition weight to the weight map. Update weight mapping collections (some RDDs may be affected by the number of uses after their application, affecting the weight).

(5) When the sum of the size of the free memory and the replacement list is smaller than the size of the partition to be cached, the partition is not cached and the list to be replaced is cleared.

IV. THEORETICAL ANALYSIS AND EXPERIMENTS

A. Comparison of Cache Replacement Algorithm

At present, there are many weight buffer replacement algorithms for Spark has been proposed, and the weight calculation involves parameters including the frequency, size, and calculation cost of the replacement target. In the following, The new minimum weight cache replacement algorithm SWCR we proposed compared with the LWR proposed in [5] and the cache replacement algorithm LRU used in Spark, as shown in Table 2.

Table 2 Comparison of related cache replacement algorithms

algorithm / parameter	LWR	LRU	SWCR
replace target	RDD	RDD partition	RDD partition
usage frequency	Yes	No	Yes
Partition size	No	No	Yes
Computing cost	Yes	No	Yes
Dependency integrity	No	No	Yes
Correction parameter	Yes	No	Yes

LWR's calculation of weights is based on the assumption that two RDDs with the same frequency of use and equal computational cost are not present in the task, so the influence factor of the RDD partition size is ignore. Compared with the SWCR proposed in this paper, the weight influence factors considered by LWR are not comprehensive enough, and in the case of insufficient memory, the RDD to be replaced by the weight is determined, and the replacement of the entire RDD may affect the execution of other tasks. SWCR not only considers the size of the RDD partition, but also replaces the target with an RDD partition. It is more accurate than the LWR in the replacement partition selection.

LRU ignores Spark's data characteristics when performing memory replacement, considers only the time factor in which RDD is accessed in memory. And when the RDD partition with high reutilization rate or high recalculation cost is replaced because it has not been used recently, it will cause unnecessary computational costs and affect application execution efficiency. The SWCR replacement algorithm in this paper comprehensively considers several factors affecting RDD partitions. The linear weighted accumulation method is used to construct the weight calculation model. The weight value comparison can replace the relatively insignificant RDD partition more accurately, so that the memory can be fully obtained, thereby improve application execution efficiency and improve Spark computing performance.

B. Experimental design and results evaluation

Setting up a Spark cluster to verify the effectiveness of memory cache optimization in this article.

Set up an experimental environment on a server Think Server whose operating system is Ubuntu14.4, create 4

virtual machines on the server, use these four virtual machines to build a Spark cluster, one of which is the master node, and the other three as worker nodes. Use Spark 2.4.0 as a parallel computing processing framework, Hadoop 3.7 and Hadoop yarn are used as resource scheduling modules. Use PageRank as the task algorithm, because PageRank algorithm is a typical data-intensive algorithm, it will involve multiple iterations. And it will effectively improve the efficiency of calculation when using cache. The experimental data was selected from the standard dataset provided by SNAP [15], from which two datasets were selected. The dataset details are shown in Table 3.

Table 3 SNAP data set

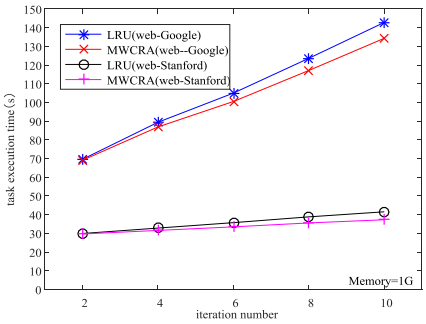
Name	Nodes	Edges	Description
web-Google	875,713	5,105,039	Web graph from Google
web-Stanford	281,903	2,312,497	Web graph of Stanford.edu

(1) Algorithm verification

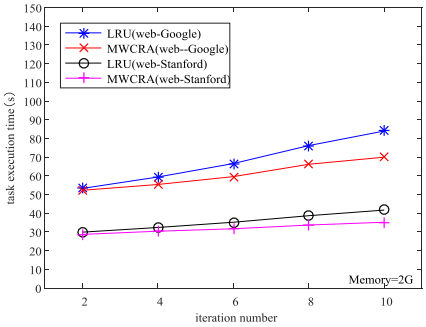
In order to verify SWCR, we implemented SWCR on the Spark platform. The initial weight values for each factor of SMWCR are A= {a, b, c, d} = {0.4, 0.3, 0.2, 0.1}. In order to contrast with the SWCR proposed in this paper, Spark's own cache replacement algorithm LRU is selected for comparison.

We select two data sets with large differences in size to test the PageRank algorithm in the experiment. Use Spark's default cache replacement algorithm to record the results of different iterations. The number of iterations is ten-fold cross-validation. The running time of each algorithm is recorded. The average value calculated is the execution time of the iterations. Then use the new weight buffer replacement algorithm for the same experiment, averaging the execution time of the records. The number of iterations is 2, 4, 6, 8, 10 respectively. Experiments were carried out under the conditions that the executor memory was 1G, 2G, and 3G. The experimental results are compared as shown in Fig.4.1

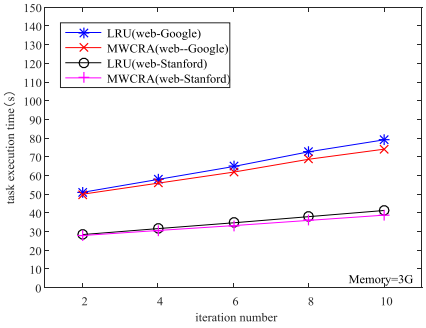
Fig.4.1 shows a comparison of execution times with LRU and SWCR in different memory configurations. Under the same data set, SWCR can effectively reduce execution time relative to LRU. Under the same memory condition, when the data set is larger, the efficiency of SWCR to improve the efficiency of PageRank is more obvious. Comparing the Fig4.1 (a), (b), and (c), we can see when the memory is 1G, its performance is relatively poor because it takes up memory for the calculation of weights. The larger the memory, the better the performance. When the memory is large enough, the advantages of the minimum weight cache replacement algorithm are no longer significant.



(a) Comparison of task execution time when memory is 1G



(b) Comparison of task execution time when memory is 2G



(c) Comparison of task execution time when memory is 3G

Fig.4.1 LRU and SMWCR execution time comparison in different memory configurations

(2) Performance evaluation

The effectiveness of the SWCR in this paper is verified by comparing the task execution speedup. Executor memory is 2G, calculate and record the task execution speedup according to formula (8), as shown in Fig.4.2. Compared with the Spark task execution speedup comparison using the LWR [5], it can be seen that SWCR optimizes the Spark memory cache better than LWR, and SWCR can further improve the Spark computing performance.

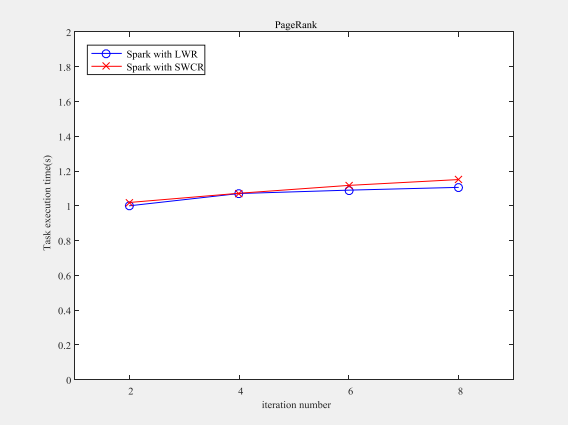


Fig.4.2 Spark computing performance evaluation comparison

V. CONCLUSION

In Spark, when there is insufficient storage memory, Spark adopts the LRU to select the Block to be replaced. It does not consider the RDD partition influencing factors. In this paper, we use the weight replacement algorithm and improve it. Dependency integrity is added to the minimum weight cache replacement algorithm, and the value of each partition feature can be dynamically adjusted according to the task execution condition, which increases the adaptability for cache replacement and improves the task execution efficiency under the memory bottleneck. The comprehensive evaluation proves that the replacement algorithm this paper proposed can effectively improves the performance of Spark computing.

REFERENCES

[1] Zaharia M, Das T, Li H, et a1 · Discretized streams： An efficient and fault-tolerant model for stream processing on large clusters [A] · Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing[C] · Boston，MA：USEN／X，2012, pp.1-6 ·

[2] Zaharia M, Chowdhury M, Franklin M, et al. Spark: cluster computing with working sets [C]. Usenix Conference on Hot Topics in Cloud Computing, 2010, 15(1) ：pp.10-10.

[3] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets ： a fault-tolerant abstraction for in-memory cluster computing[C]//Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation，2012,pp.1-14

[4] Duan M , Li K , Tang Z , et al. Selection and replacement algorithms for memory performance improvement in Spark[J]. Concurrency and Computation: Practice and Experience, 2016, 28(8):2473-2486.

[5] Bian, C.; Yu, J.; Ying, C. T.; Xiu, W. R. (2017): Self-adaptive strategy for cache management in spark. Acta Electronica Sinica, vol. 45, no. 2, pp. 278-284.

[6] Jiang ZP, Chen HP, Zhou H, et al. An elastic data persisting solution with high performance for spark[C]. IEEE International Conference on Smart City/ Socialcom/ Sustaincom, 2016:pp. 656-661.

[7] Yu Y, Wang W, Zhang J, et al. LRC: Dependency-aware cache management for data analytics clusters[C]// IEEE INFOCOM 2017 - IEEE Conference on Computer Communications. IEEE, 2017.

[8] Yu Y, Wang W, Zhang J, et al. LERC: Coordinated Cache Management for Data-Parallel Systems[C]// GLOBECOM 2017 - 2017 IEEE Global Communications Conference. IEEE, 2017.

[9] Ho L Y , Wu J J , Liu P , et al. Efficient Cache Update for In-Memory Cluster Computing with Spark[C]// IEEE/ ACM International Symposium on Cluster. IEEE, 2017.21-30

[10] Swain D, Paikaray B, Swain D. AWRP: Adaptive Weight Ranking Policy for Improving Cache Performance [J]. Computer Science, 2011:209-214

[11] CHEN K, WANG B, FENG L. Data Object Cache in Spark Computing Engine [J]. ZTE Technology Journal, 2016, 22(2):pp.23-27.

[12] Meng, H. T.; Yu, S. P.; Liu, F.; Xiao, N. (2017): Research on memory management and cache replacement policies in spark. Computer Science, vol. 44, no. 6, pp. 31-35

[13] LIU H, TAN L. New RDD Partition Weight Cache Replacement Algorithm in Spark [J]. Journal of Chinese Mini-Micro Computer Systems, 2018, 39(10):pp.153-158.

[14] Ananthanarayanan G, Ghodsi A, Wang A, et al. PACMan: Coordinated Memory Caching for Parallel Jobs [J]. Usenix Nsdi, 2012:pp.20-20.

[15] SNAP (2019) : Stanford network analysis project. http://snap.stanford.edu/data/