 CMC, vol.56, no.3, pp.415-431, 2018

An Improved Memory Cache Management Study Based on Spark

Suzhen Wang[1], Yanpiao Zhang[1], Lu Zhang[1], Ning Cao[2,*] and Chaoyi Pang[3]

Abstract: Spark is a fast unified analysis engine for big data and machine learning, in which the memory is a crucial resource. Resilient Distribution Datasets (RDDs) are parallel data structures that allow users explicitly persist intermediate results in memory or on disk, and each one can be divided into several partitions. During task execution, Spark automatically monitors cache usage on each node. And when there is a RDD that needs to be stored in the cache where the space is insufficient, the system would drop out old data partitions in a least recently used (LRU) fashion to release more space. However, there is no mechanism specifically for caching RDD in Spark, and the dependency of RDDs and the need for future stages are not been taken into consideration with LRU. In this paper, we propose the optimization approach for RDDs cache and LRU based on the features of partitions, which includes three parts: the prediction mechanism for persistence, the weight model by using the entropy method, and the update mechanism of weight and memory based on RDDs partition feature. Finally, through the verification on the spark platform, the experiment results show that our strategy can effectively reduce the time in performing and improve the memory usage.

Keywords: Resilient distribution datasets, update mechanism, weight mode.

1 Introduction

Human society has entered the era of Big Data which has become one of the most important factors in production. The big data of industry and enterprise that can reach hundreds of TB or even hundreds of PB at a time has far exceeded the processing capacity of traditional computing techniques and information system. Cloud computing enables convenient and on-demand access to a shared pool of configurable computing resources, see Li et al. [Li and Liu (2017); Liu and Xiao (2016); Zhao, Wang, Xu et al. (2015); Thanapal and Nishanthi (2013)]. Spark has been widely adopted for large-scale data analysis, see Apache Spark [Apache Spark (2018)]. One of the most important capabilities in Spark is persisting (or caching) a dataset in memory or on disk across operations, see Lin et al. [Lin, Wang and Wu (2014); Zaharia, Chowdhury, Franklin et al. (2010)]. However, the user manually selects the RDD to cache by experience, which leads to several uncertainties and impact on efficiency. At the same time, the Spark can automatically monitor cache usage on each node and drop out old data partitions in a LRU fashion.

[1] Hebei University of Economics and Business, Shijiazhuang, Hebei, 050061, China.

[2] University College Dublin, Belfield, Dublin 4, Ireland.

[3] The Australian e-Health Research Centre, ICT Centre, CSIRO, Australia.

[*] Corresponding Author: Ning Cao. Email: 343412081@qq.com.

However, in Spark, the LRU algorithm only considers the time feature of the node where the partition is located but not the partition features.

Recognizing this problem, scholars have researched various replacement algorithms. By studying FIFO, LRU and other cache algorithms, the AWRP algorithm that calculates the weight for each object is designed in Swain et al. [Swain and Paikaray (2011)], but it assumes that the size of the blocks is equal. For the Spark framework, Bian et al. [Bian, Yu, Ying et al. (2017)] propose an adaptive cache management strategy from three aspects: automatic selection algorithm, parallel cache cleanup algorithm and lowest weight replacement algorithm. But it ignores the factor that the weight is changing during execution. The WR algorithm in Duan et al. [Duan, Li, Tang et al. (2016)] is proposed to calculate the weight of RDDs by considering the partitions computation cost, the number of use for partitions, and the sizes of the partitions. However, it does not take into account the change in the remainder usage frequency of the cache partition during task running. Before the persistence, Jiang et al. [Jiang, Chen, Zhou et al. (2016)] further consider whether the persistence is needed by judging the cost required and the computing cost. Zhang et al. [Zhang, Shou, Xu et al. (2017); Chen, Zhang, Shou et al. (2017); Chen and Zhang (2017)] put forward fine-grained RDD check-pointing and kick-out selection strategy according to DAG diagram, which effectively reduces RDD computing cost and maximizes memory utilization rate. Meng et al. [Meng, Yu, Liu et al. (2017)] taking full account of the distributed storage characteristics of RDD partition point out the influence of complete RDD partition and incomplete RDD partition on cache memory, but he does not discuss the choice of RDD caches.

By analyzing the researches above, in this paper, we optimize the selection strategy for caching RDD and the LRU replacement algorithm from the following aspects:

(1) We propose a prediction mechanism for caching RDDs. The use frequency of each RDD is obtained based on the DAG diagram, and it is further decided whether to cache the RDD according to the cost between the persistence and computation.

(2) We put forward a weight replacement algorithm based on RDD partition features According to the characteristics of partition, we calculate the weight model by using entropy method.

(3) The update mechanism for storage memory and weight of partition. Whenever the usage frequency of the partition changes, we update the partition weights and the storage memory in real time.

2 Introduction of Spark cache

2.1 Resilient distribution datasets

RDD is distributed collection of objects, that is, each one can be divided into multiple partitions. RDDs support four types of operations: Creations, transformations, controls and actions. The SparkContext is responsible for the creation of RDD. Additionally, the transformation operations create a new dataset from an existing one, and the control operations mainly persist the RDD. In the course of the execution of the program, the operation of producing RDD is delayed until the action operation happened, see Ho et al. [Ho, Wu, Liu et al. (2017)].

During the task execution, a DGA graph based on the Lineage can be created by DAGSheduler, and further divides the stage based on the dependency between the RDDs. See Gounaris et al. [Gounaris, Kougka, Tous et al. (2017); Geng (2015)] for more details. Each stage creates a batch of tasks which are then assigned to various executor processes. After all the tasks in a stage are executed, the reused RDD would be stored in the cache for further use, as illustrated in Fig. 1. The reused data is more common in some iterative computations, such as the PageRank and *K*-means mentioned in Zaharia et al. [Zaharia, Chowdhury, Das et al. (2012); Xu, Li, Zhang et al. (2016); Zhang, Shou, Xu et al. (2017); Chen, Zhang (2017); Napoleon and Lakshmi (2010)]. It can effectively reduce the cost of computing by caching the reused RDD to the storage memory. During the execution, users can specify the caching level and the object to be stored, such as the MEMORY_ONLY and MEMORY_ONLY_2 mean to store the RDD to the memory [Ding and He (2004)].

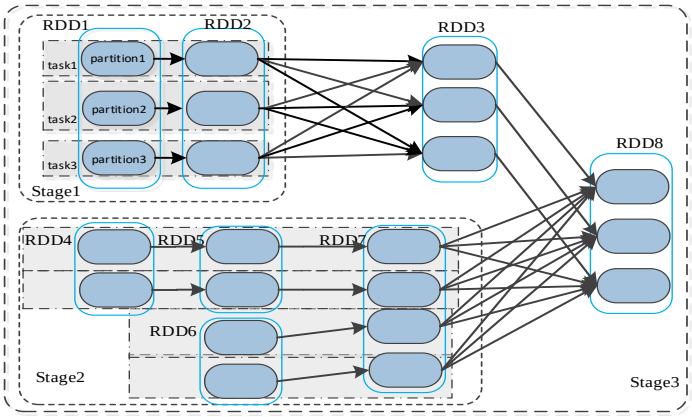


Figure 1: The schematic diagram of parallel computing for RDD

2.2 Memory management mode

As shown in Fig. 2, the schematic diagram of memory partition in Spark 2.0.1 is obtained by analyzing the source code, see Dabokele et al. [Dabokele (2016); Hero1122 (2017)]. The memory is firstly divided into two main parts: Memoryoverhead (default 384 M) and ExecutorMemory. ExecutorMemory can be further divided into Reserved Memory (default 300 M) and UsableMemory. If the system memory is less than 1.5* ReservedMemory, there would be an abnormality report. Note that 60% (the proportions can be modified) of the UsableMemory is used for storage and calculation. HeapExecutorMemory is used for task computing, and HeapStorageMemory is mainly used to cache the intermediate results that need to be reused. After Spark 1.6, the StorageMemory and the ExecutorMemory can be dynamically converted to each other, what is called Unified. Stroage and Execution can borrow each other's memory. It is important to note that when there is no enough memory in both space, the Storage portion will spill the data that over 50% to the disk (based on the storage_level) until the memory borrowed is returned, this is because the execution (Execution) is more important than a cache (Storage). There the release data is also based on the LRU algorithm.

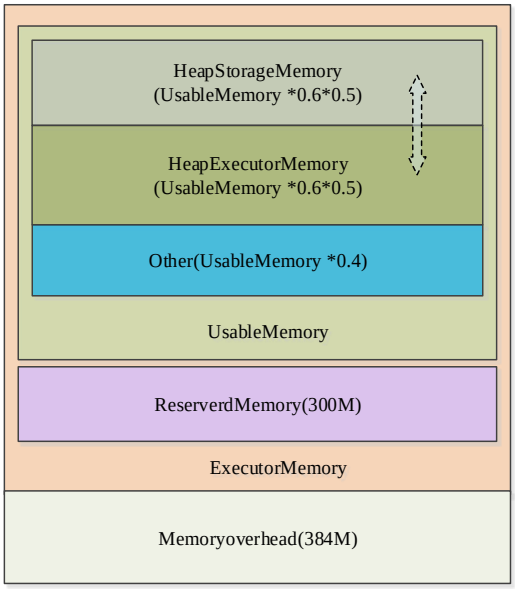


Figure 2: The diagram of Spark memory allocation

2.3 Cache mechanism in Spark

All of the calculations in Spark are done in the memory, and when there is a RDD that needs to be reused, it would be cached by experience. When storage capacity is insufficient, something essential needs to be done with replacement algorithm to reclaim the memory. The default algorithm is LRU which is mentioned in He et al. [He, Kosa and Scott (2007)]. The principle of LRU algorithm is: (1) The data newly added is inserted into the head of the linked list. (2) The data accessed is moved to the chain header. (3) When the storage space of the linked list is insufficient, the data at the end of the linked list is discarded. As shown in Fig. 3.

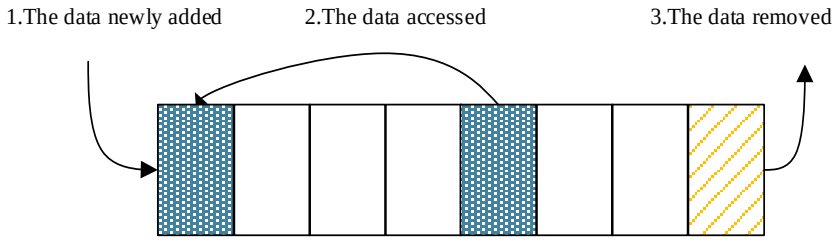


Figure 3: The principle of LRU algorithm

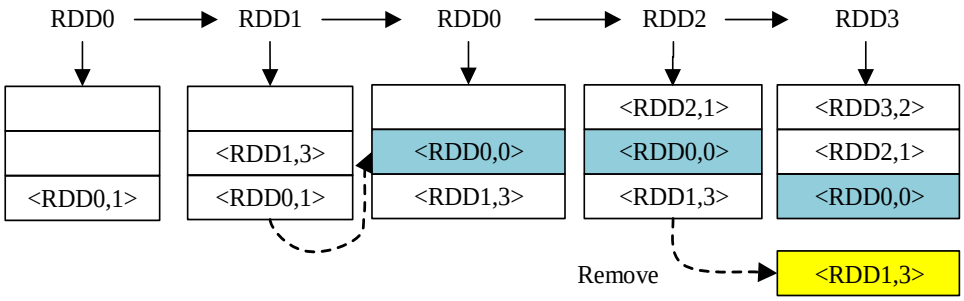


Figure 4: The cache replacement schematic of RDD

In Spark, the LRU replacement algorithm is implemented by LinkedHashMap with the characteristics of a double linked list, see Lan [Lan (2013)]. Because of its inability to predict the future use of each page, it will release the least recently used page, see Wang [Wang (2014)]. However, in Spark, different RDD partitions in the same storage memory are heterogeneous, that is, they are different in the size and usage frequency. In this case, it would lead to a lot of unnecessary calculations only by the time factor.

For instance, let RDD and the corresponding usage frequency represent for the series of RDDs: <RDD0, 2>, <RDD1, 4>, <RDD2, 2>, <RDD3, 3>. The Fig. 4 shows the cache replacement schematic of RDD. While the RDD is cached, the usage frequency decreases by one. In accordance with the ideology of LRU, when RDD0 is used for the second time, it will be placed in front of RDD1 along with the frequency of RDD0 drops to zero, which means the RDD0 would not be reused in future operations. When RDD3 will be cached, the RDD1 with frequence of 3 would be released because of the insufficient memory. That is we will recalculate the partition next time, which would make unnecessary computational cost. From above we can see that when a RDD partition does not need to be reused in the next calculation, it may still occupy memory space. While when a RDD may be reused next time, it may have been freed from memory. The LRU algorithm is also used when the ExecutorMemory is not enough, and the StorageMemory would put the more valuable partition data to the disk or somewhere else. Neither cache replacement nor memory recall can meet the demand of task computing well, so it is necessary to develop the replacement strategy based on partitioning features of RDD.

3 Cache replacement model of RDD

In this section, we will learn from the follow parts. Firstly, analyze the influence factors of RDD cache, and then propose three innovations: The prediction mechanism for persistence, the weight model by using the entropy method, and the update mechanism of weight and memory based on RDDs partition feature.

Note that each job contains several RDDs, now let $R = \{R_1, ...R_i, ...R_n\}$ be the set of RDDs, while $RP = \{RP_{i1}, ...RP_{ij}, ...RP_{im}\}$ be the set of partitions of R_i.

Definition 1 Task execution speedup. There we use the task execution speedup expressed by TE_{sp} to measure the task performance with the optimized algorithm. There will be better performance of the task execution with the greater speedup. The Formula is as follows:

$$TE_{sp} = \frac{T_{LRU}}{T_{opt}} \tag{1}$$

Set T_{LRU} as the execution time with LRU algorithm, and the T_{opt} as the execution time with optimized algorithm.

3.1 Analysis of influence factors

To improve the research, it is necessary to learn the characteristics of RDD. The characteristic elements are as follows:

(1) The frequency of utilization

In order to avoid the unnecessary computation, it is necessary to make a judgment on the usage frequency of RDD. When an action occurs, the DAGScheduler creates a DAG based on the Lineage of RDD. By traversing the DAG diagram, we use $G < R_i, N_i >$ to represent the characteristics of the RDD, in which the N_i represents the total usage frequency during the entire program. The RDD with larger N_i is more worthy of being cached.

(2) The remainder use frequency of RDD partition

The RDD caching is in the form of partitions, and the residual frequency of the partitions decreases gradually in the course of task execution. There let N_{ij} be the usage frequency for each RDD partition. Before caching the R_i, we set the equation: N_i=N_{ij}. When the first caching occurs of R_i, the value of the N_{ij} is reduced by one, and also it continues to decrease whenever the R_i is used.

(3) Computational cost

When the cache memory is insufficient, the LRU algorithm will release the least recently used RDD. In the system, the algorithm only takes into account the time feature of the node where the partition is located. In fact, there would be unnecessary computational overhead provided that the partition eliminated needs to be reused next time. Therefore, the computational cost of partition should be a crucial factor. The partition with higher cost shouldn't be replaced. Here we use $T_{RP_{ij}}$ defined in Duan et al. [Duan, Li, Tang et al. (2016)] to express the computational cost of RDD partition.

$$T_{RP_{ij}} = ET_{ij} - ST_{ij} \tag{2}$$

Let ET_{ij} represent for the finish time, while the ST_{ij} as the start time. At the same time, the execution time of a RDD is determined by the maximum time of all partitions, so the computational cost of RDD is as follows:

$$T_{R_i} = \max\{T_{RP_{i1,}}, ..., T_{RP_{ij.}}, ..., T_{RP_{in}}\} \tag{3}$$

(4) The size of partition

The partitions that occupy the larger memory space should be preferentially eliminated to release more resources.

3.2 The prediction mechanism for caching

The prediction mechanism is divided into two parts:

(1) When N_i is equal to 2

In this case, the frequency will change to 1 by storing RDD in the cache. If not cache, there would be a recomputation cost. So it is necessary to decide whether it is worth to cache the RDD according to the relation:

$$\frac{S_{R_i}}{V_{R_i}} < T_{R_i} \tag{4}$$

Where S_{R_i} is the size of a RDD, and V_{R_i} is the speed for data persisting. If the re-computing cost is larger than caching cost, it is suggested to cache the RDD.

(2) When N_i>2, it is suggested to be cached

The execution process is as follows:

Algorithm 1: RDD automatic cache prediction algorithm.

```
Input：RDD sequence: R={R1...Ri...Rn}
       the usage frequency of RDD: NR
       the partition of RDD: RP
       the size of RDD partition: SRP
       the frequency to be used of RDD partition: NRP
       the remaining memory size of the storage node: Scach
       the C is donated as the set of partitions cached.
Initialization：NR=NRP
For (i=1 to n)
  If (NR=2 and SRP<Scach)
      if (cacheCost<countCost) then
         C=C ∩ RP
         NRP=NRP-1
      end if
   else if (NR>2 and SRP<Scach)
      C=C ∩ RP
      NRP=NRP-1
   else if (SRP>Scach)
      call Algorithm 2
   end if
end for
```

3.3 Weight replacement model

Replacement operations are required when the storage space is insufficient to accommodate the RDD that needs to be cached. Also, the Storage section can apply to all free memory in the Execution section. When the execution requires more memory, the storage portion will spill the data to the disk (based on the storage_level) until the

memory borrowed is returned. Forced discard data is also based on LRU algorithm. In order to represent the importance of an index in the whole analysis process, we adopt the weight form. Therefore, to reduce the cost of the re-computation, we put forward the weight calculation model based on partition feature by using entropy method in Zuo et al. [Zuo, Cao and Dong (2013)] to optimize LRU algorithm. The entropy method determines the index weight based on the degree of variation of each index value. Entropy is a measure of the degree of disorder in the system and can be used to measure the effective information of known data packets and determine weights. By determining the weights based on the calculation of the entropy, the weight of each partition is determined according to the degree of difference in the feature values of the partitions. When there is a large difference between certain eigenvalues of the evaluation object, the entropy is smaller which means the number of valid information provided by this feature is larger. Accordingly, the weight of the object should be larger. Conversely, if the difference between a certain feature value is small, the entropy value should be larger. It indicates that the feature provides less information and the weight should be smaller. When a feature value of a partition is exactly the same, the entropy value reaches a maximum, which means that the feature value is useless information. The process of the weight calculation is as follows:

(1) Converting the characteristic of partition into matrix form

Assuming that there are n partitions in the storage memory, and each partition has m feature attributes. In this paper we set m=3, which means there are three features: N_{ij} , $T_{RP_{ij}}$ and SR_{ij}. Let X_{ij} be the value of the j-th index of the i-th partition, the matrix is as follows:

$$RP = \begin{pmatrix} X_{11} ,..., X_{1m} \\ . \quad\quad . \\ . \quad\quad . \\ . \quad\quad . \\ X_{n1} ,..., X_{nm} \end{pmatrix}_{n\times m} \tag{5}$$

(2) Normalization processing

Since the measurement units of each feature are inconsistent, the standardized operations must be performed before computing. That is, the absolute value of the eigenvalue should be converted to the relative value to solve the homogeneity problem of different eigenvalues. The N_{ij} and $T_{RP_{ij}}$ are positive correlation index, and SR_{ij} belongs to negative correlation index. The normalized treatment formula is as follows:

$$X_{ij}' = \begin{cases} \dfrac{X_{ij} - \min\{X_{ij},..., X_{nj}\}}{\max\{X_{1j},..., X_{nj}\} - \min\{X_{1j},..., X_{nj}\}} & , \text{ Positive indicators} \\ \dfrac{\max\{X_{ij},..., X_{nj}\} - X_{ij}}{\max\{X_{1j},..., X_{nj}\} - \min\{X_{1j},..., X_{nj}\}} & , \text{ Negative indicators} \end{cases} \tag{6}$$

For convenience, the normalized data X'_{ij} is still represented by X_{ij}.

(3) Under the j-th feature, the i-th partition occupies the proportion of the feature

$$P_{ij} = \frac{X_{ij}}{\sum_{i=1}^{n} X_{ij}} \qquad (j = 1,2....m) \tag{7}$$

(4) The entropy of the *j*-th feature

$$e_j = -k\sum_{i=1}^{n} P_{ij}\ln(P_{ij}) \tag{8}$$

Where the $k>0$, ln *i*s natural logarithm, then the $e_j \geq 0$, and $k=1/\ln m^2$.

(5) The difference coefficient of the *j*-th eigenvalue

$$g_j = 1 - e_j \tag{9}$$

(6) The weight of each feature

$$W_j = \frac{g_j}{\sum_{j=1}^{m} g_j} \qquad (j = 1,2...m) \tag{10}$$

(7) The weight of each partition

$$V_i = \sum_{j=1}^{m} W_j * P_{ij} \qquad (i = 1,2....n) \tag{11}$$

Finally, by calculating the weight of each partition, the partition with lowest weight should be considered first to be replaced when the replacement happened. The process is as follows:

Firstly, we need to compare the weight of the partition to be cached with the lowest weight:

(1) If there is a qualified partition in the cache, release the partition. Otherwise, conversely turn to (2).

(2) Put the RDD which needs to be cached to the wait cache area, and wait for weight updating in the storage memory.

The execution process is as follows:

Algorithm 2: weight replacement

Input： partition in cache area: *Rp*, weight： *v*
the size of *Rp*： S_{RP}
surplus space of storage memory： S_{cach}
the *C* is donated as the set of partitions cached
the set of the size for partitions cached: $\{CS_{RP1} \ldots CS_{RPp}\}$
for (*i*=1 to *p*)
if (C_{RPi}.weigth < *v*)
put C_{RPi} to weigthList[*j*] according to the weight
from small to large
end if
end for
for (*i*=1 to weigthList.length)
if ($SRP<Scach-CS_{RPi}$)
Rp replace C_{RPi}
$N_{RP=} N_{RP}$-1
else if
C=*C*
waiting for weight update
end if
end for

3.4 The update mechanism for storage memory and weight of partition

As we can see that in the process of task execution, the frequency of each partition to be used is constantly decreasing with the use of partitions, and the weights of the corresponding partitions are also changing. So we propose the update mechanism for storage memory and weight of partition: Whenever a partition in a storage area is used, the usage frequency of the partition is reduced by one, and all partitions are traversed to update the weight. At the same time, the partition to be used with frequency zero should be released to release more memory. During task execution, the weight of the partition should be updated whenever the remainder usage frequency of partitions in the storage area is reduced. The execution process is as follows:

Algorithm 3: Update mechanism

Input：the remainder usage frequency of partitions：N_{RP}
the C is donated as the set of partitions cached: $C=\{C_{RP1},...,C_{RPi},...,C_{RPp}\}$
while a RDD will be used in computing:
for (i=1 to p)
if (C_{RPi} will be used) then
$N_{RPi=}N_{RPi}$ -1
renew C_{RPi}.weigth
end if
if (N_{RPi}==0)
$C=C-C_{RPi}$
end if
end for

4 Experimental verification

The environment required in this experiment is as follows:

(1) Cluster environment: Six virtual machines that created by two laptop computers and a desktop.

(2) Cloud environment: Use Spark 2.0.1 as the computing framework and Hadoop Yarn as resource scheduling module.

(3) Monitoring environment: nmon and nmon analyser.

(4) Use The PageRank and *K*-means as the task algorithm, and choose three datasets from SNAP et al. [SNAP (2018)] and [UCI (2007)] respectively. The datasets are shown in Tab. 1 and Tab. 2.

Table 1: The Description of datasets from SNAP

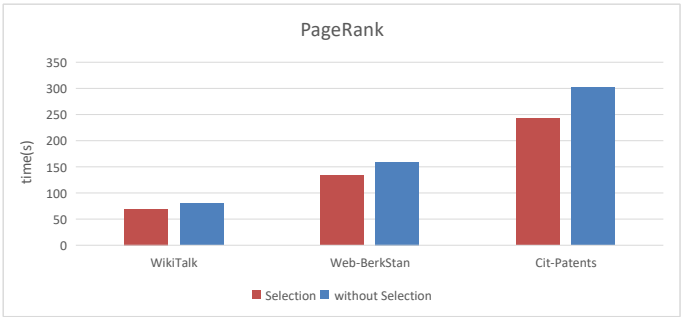
Name	Size	Description
WikiTalk	63.3 M	Communication network of Wikipedia (till January 2008)
Web-BerKStan	105 M	Web graph of Berkeley and Stanford
Cit-Patents	267 M	Citation network among US Patents

Table 2: The Description of datasets from UCI

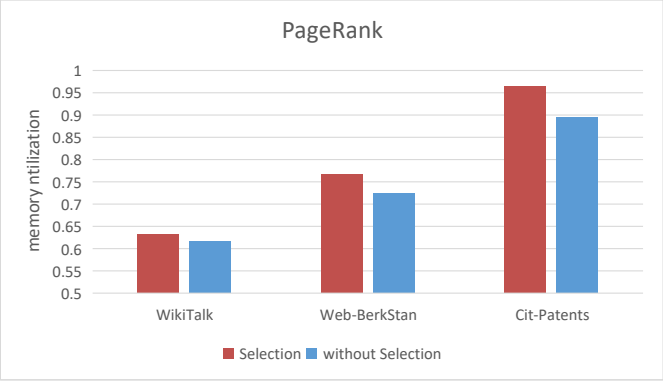
Name	Size	Description
NIPS_1987-2015	127 M	This data set contains the distribution of words in the full text of the NIPS conference papers published from 1987 to 2015.
LD2011_2014	249 M	This data set contains electricity consumption of 370 points/ clients
USCensus1990	334 M	The USCensus1990raw data set contains a one percent sample of the Public Use Microdata Samples (PUMS) person records drawn from the full 1990 census sample.

4.1 The verification for RDD cache prediction mechanism

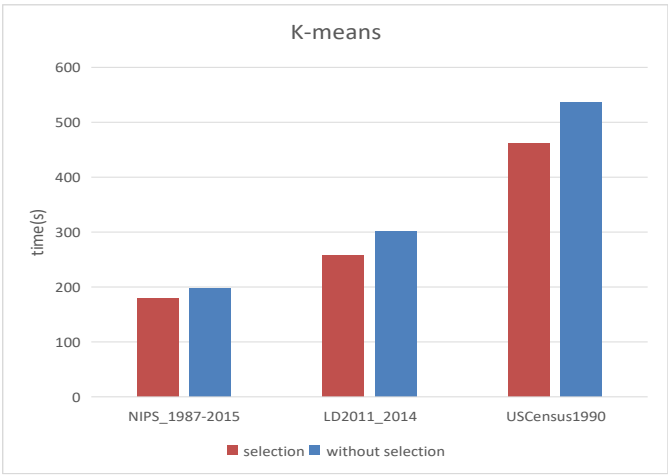
This experiment was carried out under the PageRank and K-means tasks respectively, meanwhile completed under different sizes of data sets. In this experiment, we mainly compare and analyze the difference in execution time and memory utilization rate under without RDD cache selection and with RDD cache selection. As shown in Fig. 5, the results of each experiment are obtained by running 5 times. The Figs. 5(a) and 5(b) show the experimental results under the PageRank task, while the Figs. 5(b) and 5(c) are under the K-means. Through the comprehensive analysis of Figs. 5(a) and 5(c), when the dataset is relatively small, the task execution time is short, so the difference is not obvious. Along with the amount of data increases, the performance with the prediction mechanism for caching is well. Through the comprehensive analysis of Figs. 5(b) and 5(d), the memory usage of the task with the prediction mechanism is very high mainly because the cache data will occupy storage memory after the cache mechanism is optimized. In summary, the prediction mechanism for caching reduce the execution time and improve the rate of memory usage to a certain extent.



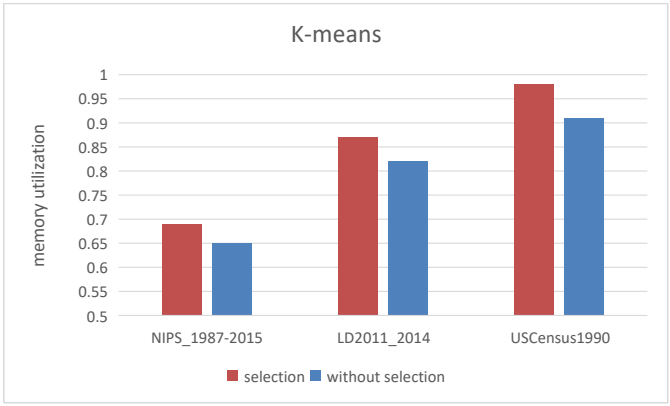
(a) Time comparison under PageRank task



(b) Comparison of memory utilization under PageRank task



(c) Time comparison under *K*-means task



(d) Comparison of memory utilization under *K*-means task

Figure 5: Schematic diagram of task execution time and memory utilization under optimization and unoptimized cache mechanism

4.2 The verification for weight replacement algorithm

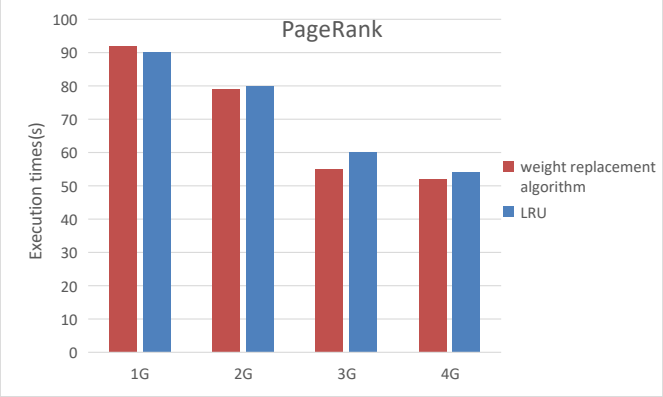
This part is implemented by modifying the *evictBlocksToFreeSpace* function in the source file named *MemoryStore*.

The datasets of WikiTalk and NIPS_1987-2015 have been used in this verification separately. To ensure that there are multiple RDDs to be cached, the dataset is divided into several small datasets separately in the process of data reading by analyzing implementation code of PangRank and K-means. Now this process satisfies the conditions for multiple RDDs to be cached.

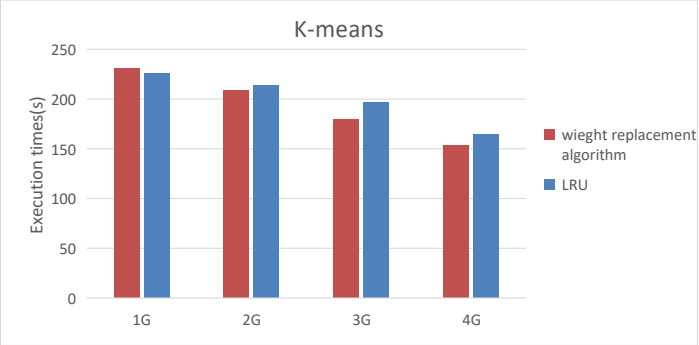
Another condition is that there are RDDs that need to be cached, so we verify the weight replacement algorithm by using the RDD cache prediction mechanism proposed by this paper.

This algorithm is called only when the storage memory is insufficient. So this experiment is verified under different memory sizes: 1 G, 2 G, 3 G and 4 G. As shown in Fig. 6, compared with LRU algorithm, when the memory is small, our improved algorithm could not reduce the execution time well for the reason that the analyzing of the partition features and the updating for weight and memory could occupy much time. With the increase of memory, the weight replacement algorithm performs well. When the memory is large, there is enough memory to store the cache partition, and the number of the partition replacement is reduced. As we can see that the execution time of the two algorithms is similar with enough memory.

As Fig. 7 shown, we use the task execution speedup to measure the task performance. According to the Formula (1), it respectively shows the speedup under PageRank with WikiTalk and *K*-means with NIPS_1987-2015. As we can see, the optimized algorithm shows the poor performance in 1 G memory. While with the larger memory, it shows good performance. When the memory is large enough, the optimization algorithm advantage is no longer obvious.



(a) The contrast validation with WikiTalk dataset



(b) The contrast validation with NIPS_1987-2015 dataset

Figure 6: Time comparison analysis under different memory

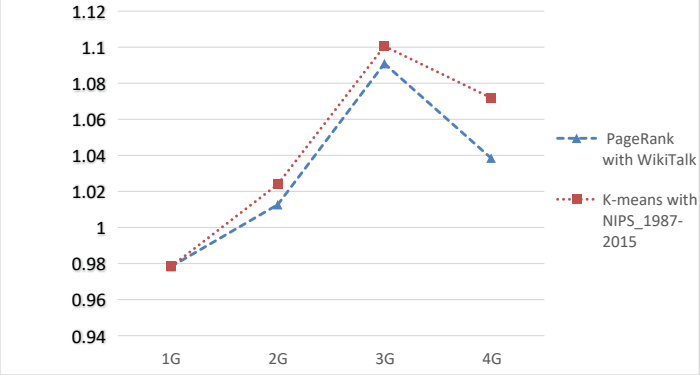


Figure 7: Task execution speedup with different tasks

5 Conclusion

The By analyzing the characteristics of the Spark RDD data model, we propose three points: (1) Proposing prediction mechanism for RDD cache through the usage frequency of RDD, and the cost of re-computation and cache. (2) Based on the entropy method, we propose a weight model based on RDD partitioning feature by analyzing the remainder frequency of RDD partition, computational cost and the size of partition. (3) Putting forward the update mechanism for storage memory and weight of partition. In the actual running scene, the memory of the cluster environment is limited. For the multitask execution with big data, this method can effectively reduce the task execution time and improve the memory utilization.

Acknowledgement: This paper is partially supported by Education technology research Foundation of the Ministry of Education (No. 2017A01020).

References

Apache Spark (2018): Apache Spark. https://spark.apache.org/.

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.

Dabokele (2016): Spark memory management-UnifiedMemoryManager and Static-MemoryManager. http://blog.csdn.net/dabokele/article/details/51475469.

Ding, C.; He, X. F. (2004): Cluster structure of k-means clustering via principal component analysis. *Lecture Notes in Computer Science*, vol. 46, no. 4, pp. 414-418.

Duan, M. X.; Li, K. L.; Tang, Z.; Xiao, G. Q.; Li, K. Q. (2016): Selection and replacement algorithms for memory performance improvement in spark. *Concurrency & Computation Practice & Experience*, vol. 28, no. 8, pp. 2473-2486.

Geng, J. A. (2015): *Spark Internals Core Design and Source Code Analysis*. China Machine Press, China.

Gounaris, A.; Kougka, G.; Tous, R.; Tripiana, C.; Torres, J. (2017): Dynamic configuration of partitioning in spark applications. *IEEE Transactions on Parallel & Distributed Systems*, vol. 28, no. 7, pp. 1891-1904.

Hero1122 (2017): The memory management mechanism and implementation principle of Spark2.1. http://www.aboutyun.com/thread-21 951 -1-1.html.

Ho, L. Y.; Wu, J. J.; Liu, P. F.; Shih, C. C.; Huang, C. C. et al. (2017): Efficient cache update for in-memory cluster computing with spark. *IEEE/ACM International Symposium on Cluster*, pp. 21-30.

He, X. B. B.; Kosa, M. J.; Scott, S. L.; Engelmann, C. (2007): A unified multiple-level cache for high performance storage systems. *International Journal of High Performance Computing and Netwotking*, vol. 5, no. 1-2, pp. 97-109.

Jiang, Z. P.; Chen, H. P.; Zhou, H.; Wu, J. (2016): An elastic data persisting solution with high performance for spark. *IEEE International Conference on Smart City/ Socialcom/Sustaincom*, pp. 656-661.

Lan, Y. (2013): Implementation of LRU algorithm with LinkedHashMap. https://www.cnblogs.com/LZYY/p/3447785.html.

Li, Z.; Liu, Y. (2017): A differential game-theoretic model of auditing for data storage in cloud computing. *International Journal of Computational Science & Engineering*, vol. 14, no. 4, pp. 341-348.

Lin, X.; Wang, P.; Wu, B. (2014): Log analysis in cloud computing environment with hadoop and spark. *International Conference on Broadband Network & Multimedia Technology*, pp. 273-276.

Liu, D.; Xiao, P. (2016): An energy-efficient adaptive resource provision framework for cloud platforms. *International Journal of Computational Science & Engineering*, vol. 13, no. 4, pp. 346-354.

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.

Napoleon, D.; Lakshmi, P. G. (2010): An enhanced k-means algorithm to improve the efficiency using normal distribution data points. *International Journal of Computational Science and Engineering*, vol. 2, no. 7, pp. 2409-2413.

SNAP (2018): Stanford network analysis project. http://snap.stanford.edu/data/.

Swain, D.; Paikaray, B. (2011): AWRP: adaptive weight ranking policy for improving cache performance. *Computer Science*, vol. 3, no. 2, pp. 209-214.

Thanapal, P.; Nishanthi, S. P. (2013): Efficient parallel data processing in the cloud. *International Journal of Computational Science and Engineering*, vol. 5, no. 5, pp. 338-342.

UCI (2007): UCI Machine learning repository. http://archive.ics.uci.edu/ml/datasets.

Wang, H. (2014): Research on the realization of LRU algorithm. *Applied Mechanics & Materials*, vol. 530-531, pp. 891-894.

Xu, L.; Li, M.; Zhang, L.; Butt, A. R.; Wang, Y. D. et al. (2016): MEMTUNE: Dynamic memory management for in-memory data analytic platforms. *IEEE International Parallel and Distributed Processing Symposium*, pp. 383-392.

Zaharia, M.; Chowdhury, M.; Franklin, M. J.; Shenker, S.; Stoica, I. (2010): Spark: Cluster computing with working sets. *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, vol. 15, no. 1, pp. 10-18.

Zaharia, M.; Chowdhury, M.; Das, T.; Dave, A.; Ma, J. et al. (2012): Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. *NSDI'12 Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, vol. 70, no. 2, pp. 2-16.

Zhang, D. B.; Shou, Y. F.; Xu, J. M. (2017): An improved parallel k-means algorithm based on MapReduce. *International Journal of Embedded Systems*, vol. 9, no. 3, pp. 275-282.

Zhang, M. Y.; Chen, R. H.; Zhang, X. W.; Wang, X. (2017): Intelligent RDD management for high performance in-memory computing in spark. *International Conference on World Wide Web Companion*, pp. 873-874.

Zhao, X. F.; Wang, X.; Xu, H.; Wang, Y. L. (2015): Cloud data integrity checking protocol from lattice. *International Journal of High Performance Computing and Netwotking*, vol. 8, no. 2, pp. 167-175.

Zuo, L. Y.; Cao, Z. B.; Dong, S. B. (2013): Virtual resource evaluation model based on entropy optimized and dynamic weighted in cloud computing. *Journal of Software*, vol. 24, no. 8, pp. 1937-1946.