A Lowest Cost RDD Caching Strategy for Spark

Yuyang Wang[1, 2, a], Tianlei Zhou[1, 2, b]

[1]School of computer science and technology, Chongqing University of Posts and Telecommunications, Chongqing, 400065, China

[2]Chongqing Engineering Research Center of Mobile Internet Data Application, Chongqing, 400065, China

[a]529504153@qq.com, [b]ztl_wangyi@163.com

Keywords: RDD, Spark memory management, Memory computing, Cache strategy

Abstract: Spark abstracts intermediate results into RDD in memory and manages them with LRU strategy to improve performance. However, RDD will be reloaded in many cases because RDD for different computing tasks have different lifecycle, which incurs additional system overhead. In this paper we proposed a lowest cost replacement strategy as Spark's cache replacement strategy to eliminate this problem. This strategy preemptively evicts RDD with small weight values from memory based on the weight model. And then, in this process, we select the solution with the lowest cost to replace the RDD in memory to improve the efficiency of Spark. Finally, experiment results show that strategy we proposed can speed up the efficiency of the whole cluster.

1. Introduction

Spark is an improvement of MapReduce [1] programming model based on Hadoop [2, 3] platform. For a large number of network transfer data and disk I/O issues, Spark abstracts intermediate results as resilient distributed dataset (RDD) and computes them in memory, witch changes data exchange from disk to memory and speed up the data processing performance. And Spark is widely applied in the platform of big data processing because of its features of memory based and good versatility. Many companies use Spark cluster to process data such as Yahoo Audience Expansion, Baidu MapReduce and Tencent Social Ads [4]. Nevertheless, Spark has drawbacks. Part of the RDD cannot be cached because memory is limited and it needs to be reloaded [5] when it is used again. And Spark use LRU to manage RDD in memory without considering RDD useage which may increase the cost of reloading RDD. The cost of reloading RDD will be reduced and Spark efficiency will be improved if we make predictions [6] about RDD and keep important RDD in memory for a long time.

2. Related work

All of Spark's operations are based on RDD and each of them generates a new RDD. As shown in Figure 1, several new RDD will be generated after transformations and each of them contains several partitions. The transformed dependencies between these different RDD partitions form the DAG diagram. Then these dependencies decompose the DAG diagram into multiple stages. The transformation of the RDD partition in the stage is assigned to a computing task. After that, the Spark scheduler sends those tasks to different nodes in the cluster for execution. Finally, each node writes the final result to the disk. Spark is a memory-based computing framework, and its execution efficiency is directly affected by the cache-replacement strategy. Current Spark research focuses on memory management and caching strategies.

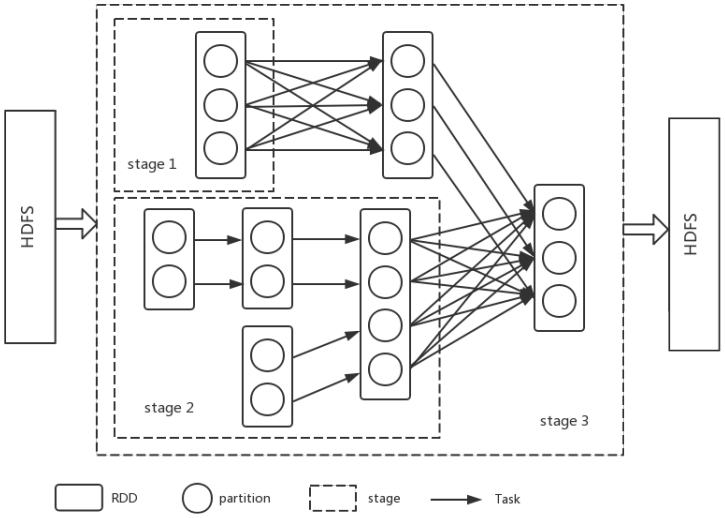


Figure 1. Spark task RDD transformation diagram

In terms of memory management. Kim [7] proposed Sparkle which replaces the current TCP/IP-based shuffle with shared memory and an off-heap memory which can update effectively. Lu [8] proposed a memory management scheme that allocates and frees memory according to the life cycle of objects rather than relying on GC (Garbage Collection) to improve the efficiency of GC.

In terms of caching strategy. Geng [9] proposed LCS (Least Cost Strategy) strategy which gets dependencies information between cache data via analyzing application, and calculates the recovery cost during running. By predicting the number of times the cached data will be reused and using it to weight the data, LCS will reduce the recovery cost by evicting data with high recovery cost. Chen [10] proposed a strategy of automatic cache RDD and the weight model of RDD, and according to this model to replace cache data. Shen [11] verified the factors influencing the weight of RDD and improved the weight model of RDD according to the experiment. They also optimized the task structure and the sequence of RDD operators in Spark task to improve Spark computing efficiency. Bian [12, 13] proposed the concept of memory resource entropy and introduced it into the RDD weight model to improve the theoretical model of RDD weight. Meng [14] proposed a strategy based on the distributed weight of RDD that evict incomplete RDD partitions firstly when memory resources are insufficient.

The above research can improve computing efficiency when Spark is short of memory resources. However, the weight-based cache replacement strategy has drawbacks. For example, multiple RDD may be replaced in memory when the RDD that needs to be cached is large, and the recovery cost of these RDD may not be minimal. The weight-based cache replacement strategy ignores the cost of RDD recovery. Therefore we adds a recovery cost model based on RDD to the weight-based cache replacement strategy, that select the scheme with the lowest recovery cost according to this model during weight-based caching replacement to reduce the cost of reload RDD.

3. Lowest cost cache replacement strategy

Spark manages RDD in memory by default with LRU strategy. Some of these RDD may be reloaded repeatedly. It is high-efficiency for spark that we predict RDD and cache RDD which reload repeatedly or reload costly. This paper proposes a lowest cost RDD cache strategy. We first establish a weight model for RDD and replace RDD in memory based on this model, then we establish recovery cost model and select the lowest cost scheme based on this model during first step.

3.1 RDD weight model

Factors that influence the weight of RDD include the frequency, size, recovery cost, and lifecycle of RDD. We define all RDD in a Spark application to be $\{RDD_1, RDD_2 \ldots, RDD_i, \ldots RDD_n\}$, and RDD_{ij} represents the jth partition of the ith RDD in the set. F_{ij} represents the usage frequency of

RDD_{ij}, S_{ij} represents the memory occupied by RDD_{ij}, LC_{ij} represents the life cycle of RDD_{ij}, $Cost_{ij}$ represents the recovery cost of RDD_{ij}, w_{ij} represents the weight value of RDD_{ij}, k represents the adjustment coefficient. Then, the weight of RDD_{ij} is calculated as Equation (1):

$$w_{ij} = \mathrm{k} \times \frac{F_{ij} \times Cost_{ij}}{S_{ij} \times LC_{ij}} \tag{1}$$

In Equation (1), the recovery cost of RDD partition is difficult to measure. The most direct way is to calculate the time of generating the RDD partition. Define the start time and finish time of generating the RDD partition task as ST_{ij} and FT_{ij}, respectively. Then, the recovery cost of RDD_{ij} can be expressed as Equation (2):

$$Cost_{ij} = FT_{ij} - ST_{ij} \tag{2}$$

3.2 RDD weight replacement algorithm

According to the weight model in the previous section, the weight value of each RDD can be calculated. Based on the weight value, the weight replacement algorithm first replaces RDD with the minimum weight value. Algorithm 1 shows the steps.

Algorithm 1 RDD Weight Replacement

Input: In-memory sets of RDD weight values RDDTreeMap, Free memory size S_{last},The RDD weight to be cached w_n, The RDD size to be cached S_n, RDD recovery costs to be cached $SCost_n$

```
1:w_m = RDDTreeMap.get(RDDTreeMap.firstKey());  //Get the minimum RDD weight in memory
2:if S_last > S_n  //There is enough memory left
3:    cache(RDD_n);
4:end if
5:if w_n < w_m  //The RDDn to be cached is smaller than the minimum weight value in the cache
6:    return;
7:else if w_n == w_m
9:    if SCost_n > SCost_m && S_last+S_m > S_n
10:       replaceRDD(RDDm,RDDn);  //Replace RDDm and RDDn in memory
11:       RDDTreeMap.remove(RDDm);  //Update the sets of records RDD
12:       RDDTreeMap.put(RDDn);
13:   else
14:       return;
15:   end if
16:else
17:   if S_last+S_m >S_n  //RDDn is larger than RDDm in memory, so it is directly replaced
18:      replaceRDD(RDDm,RDDn);
19:      RDDTreeMap.remove(RDDm);
20:      RDDTreeMap.put(RDDn);
21:   else  //Call algorithm 2 to select the optimal alternative
23: end if
24:end if
```

3.3 RDD recovery cost model

Equation (2) is the recovery cost of a single RDD partition. And an RDD partition is used multiple times in Spark applications. Therefore the recovery cost of an RDD partition is the sum of the recovery costs for the number of times that partition is used. We define F_{ij} as the usage frequency of RDD_{ij}, and $SCost_{ij}$ represents the recovery cost of RDD_{ij} in the whole process. Finally, the recovery cost of RDD partition is shown in equation 3.

$$SCost_{ij} = F_{ij} \times SCost_{ij} = F_{ij} \times (FT_{ij} - ST_{ij}) \quad (3)$$

Similarly, the recovery cost of an RDD set is the sum of all the RDD recovery costs in the set. We define set R as {RDD_m, RDD_{m+1},..., RDD_n}, and the recovery cost of set R is shown in equation 4.

$$SCost_R = \sum_{i=m}^{n} SCost_{ij} = \sum_{i=m}^{n} F_{ij} \times SCost_{ij} = \sum_{i=m}^{n} F_{ij} \times (FT_{ij} - ST_{ij}) \quad (4)$$

This paper introduces the concept of space cost entropy (Q) when we cache RDD to replace multiple small weights RDD. And we choose the RDD set that meets the condition based on the Q value. Q is used to measure the recovery cost per unit space of an RDD partition in memory. For RDD partition RDD_{ij}, its space cost entropy is the ratio between the recovery cost of the RDD and the occupied memory space, denoted as Q_{ij}, as shown in Equation 5.

$$Q_{ij} = \frac{Cost_{ij}}{S_{ij}} \quad (5)$$

The smaller the Q value is, the less expensive it is to restore the same size of RDD. If the weight replacement algorithm removes several small weights RDD from memory, we cannot guarantee that the recovery cost of these RDD is minimal. And memory may leave a lot of space. The free memory size and the sum of recovery cost are balanced when we selected RDD with the minimum Q value as the greedy strategy.

3.4 Lowest recovery cost replacement algorithm

The lowest recovery cost replacement algorithm is triggered in the process of weight replacement when the RDD to be cached is large and multiple small weights RDD may need to be replaced. Based on the recovery cost model in section 2.3, the scheme with the lowest recovery cost is selected for cache replacement. There are three possible schemes: 1. The total recovery cost of multiple small weights RDD is greater than the RDD to be cached. At this time, this RDD is ignored and not cached. We define RDD_i as the RDD to be cached, where the recovery cost is $SCost_i$ of the recovery cost of RDD_i. 2. There is an RDD in memory, whose weight is less than the RDD to be cached, and the sum of its size and free memory size can replace the RDD to be cached. At this time, the RDD to be cached only needs to replace one RDD. RDD_j is defined as RDD that meets the conditions, and the recovery cost is $SCost_j$ of RDD_j. 3. When the cached RDD replaces multiple small weight RDD, the total recovery cost of these RDD is not necessarily optimal. At this point, we use the minimum value of Q in section 2.3 as a greedy strategy to select a qualified RDD set in memory. If the set is represented by R, then the recovery cost is $SCost_R$. Algorithm 2 shows the steps.

Algorithm2 Lowest Recovery Cost Replacement Algorithm

Input:In-memory sets of RDD weight values RDDTreeMap, Sets of RDD size RDDSizeMap, Sets of RDD recovery cost RDDCostMap, Sets of RDD frequency RDDFreqMap, Free memory size S_{last}, The RDD weight to be cached w_n, The RDD size to be cached S_n

Initialization:SCost1=RDDCostMap.get(RDD_n) * RDDFreqMap.get(RDD_n); //Costs for scheme 1

SCost2=MAX_VALUE; //Costs for scheme 2

SCost3=Call Algorithm2 ; //Costs for scheme 3

1:for each item in RDDTreeMap do

2: if $w_i > w_n$

3: break;

4: end if

5: S_i = RDDSizeMap.get(RDD_i); //Size of RDD_i

6: if $S_i+S_{last} >= S_n$ //Satisfy the conditions of scheme 2

7: SCost2 = RDDCostMap.get(RDD_i) * RDDFreqMap.get(RDD_i);

8: end if

9: $Cost_i$ = RDDCostMap.get(RDD_i);

```
10:   QTreeMap.put(RDDi,Costi / Si);   //Evaluate and save Q
11: end for
12:   for each item in QTreeMap do
13:       listR.add(RDDi);   //Select RDD with the minimum Q value as greedy strategy
14:           SCost3 += RDDCostMap.get(RDDi) * RDDFreqMap.get(RDDi); //recovery cost of scheme 3
15:      SR += RDDSizeMap.get(RDDi);   //size of listR
16:      if SR + Slast >= Sn  // That satisfies the substitution condition to get the final set
17:           break;
18:      end if
19:end for
20:switch(Min(SCost1,SCost2,SCost3))
21:      case SCost1:   //Use scheme 1
22:         break;
23:      case SCost2:   //Use scheme 2
24:         //Replace the RDDn with RDDi
25:         break;
26:      case SCost3:   //Use scheme 3
27:         //Replace the RDDn with RDD in listR
28:         break;
29:end switch
```

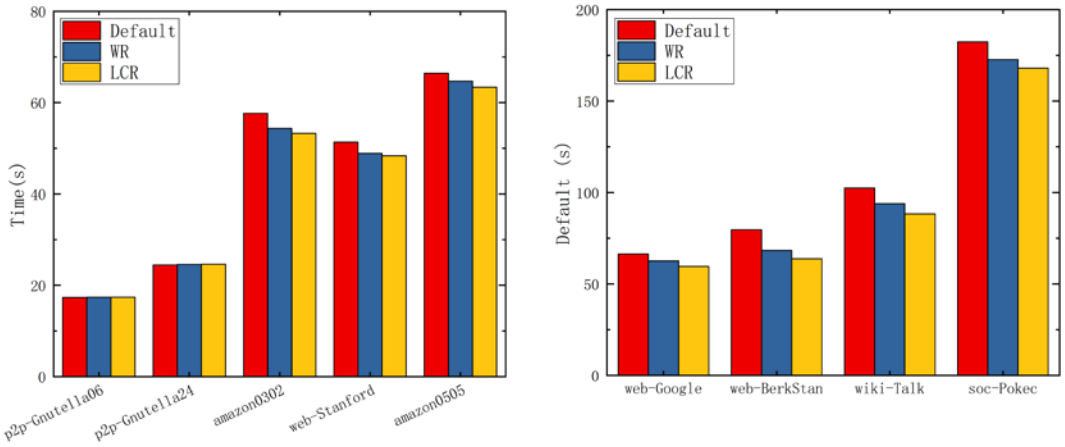
4. Experiments

This section verifies the effectiveness of weight replacement algorithm and lowest recovery cost replacement algorithm through experiments. In the experiment environment, 1 server was used as the master node and 6 servers were used as worker nodes. The data set was the PageRank test data set provided by SNAP (Stanford Network Analysis Project). And the data sets selected in this article are shown in the following table.

Table.1. SNAP partial data sets

Name	Nodes	Edges	Description
p2p-Gnutella06	8717	31525	Gnutella peer to peer network from August 5 2002
p2p-Gnutella24	26518	65369	Gnutella peer to peer network from August 24 2002
amazon0302	262111	1234877	Amazon product co-purchasing network from March 2 2003
web-Stanford	281903	2312497	Web graph of Stanford.edu
amazon0505	410236	3356824	Amazon product co-purchasing network from May 5 2003
amazon0601	403394	3387388	Amazon product co-purchasing network from June 1 2003
wiki-Talk	2394385	5021410	Wikipedia talk (communication) network
web-Google	875713	5105039	Web graph from Google
web-BerkStan	685230	7600595	Web graph of Berkeley and Stanford
soc-Pokec	1632803	30622564	Pokec online social network

Comparison under different datasets: in this section, the data set in table 1 is divided into the larger part and the smaller part. We experimented with the PageRank algorithm with 40 iterations. Figure 2 (a) and (b) compare the experiment results of native Spark's caching strategy (Default), weight replacement algorithm (WR) and lowest cost replacement strategy (LCR).



(a) (b)

Figure 2. Task execution time comparisons under different data sets

By comparing the experiment results, it can be seen that WR and LCR took shorter task execution time than Default strategy, and LCR took the shortest time. This is because RDD is often replaced when the cluster is out of memory, and WR strategy keeps important RDD in memory to reduce the time of reloading expensive RDD. LCR strategy selects the scheme with the lowest cost in the WR strategy to replace the RDD in memory to minimize the time of reload RDD.

Comparison under different iteration times: Figures 3 and 4 select amazon0601 and web-berkstan from the dataset respectively, and compare the experiment results of native Spark's caching policy (Default), weight replacement (WR) algorithm and lowest cost replacement strategy (LCR) with different iteration times.

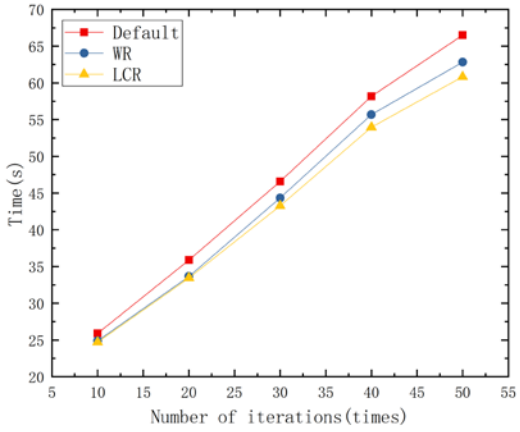


Figure 3. Task execution time in amazon0601

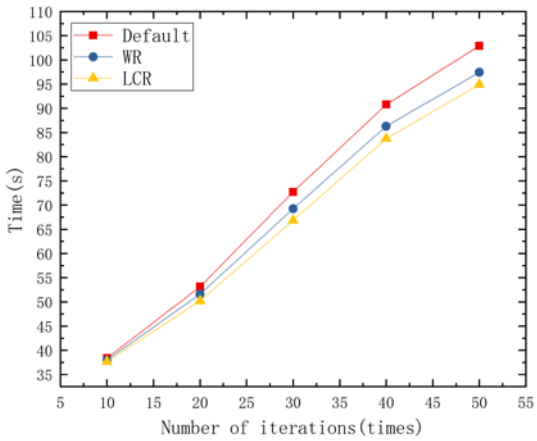


Figure 4. Task execution time in web_BerkStan

By comparing the experiment results, it can be seen that both WR and LCR performed better than Default and LCR performed best. Moreover LCR strategy performed better and better when the number of iterations was greater than 20. That is because LCR reduces the recovery cost caused by RDD replacement and improves Spark's running efficiency.

5. Conclusion

This article replaces Spark's native caching policy with a weight replacement policy. Firstly, we calculate the weight value for each RDD based on the weight model, and prioritize the elimination of RDD with small weight value. Then, in this process, we select the solution with the lowest recovery cost to replace the RDD in memory. Finally, experiment results show that the weight-based lowest-cost cache replacement strategy improves Spark's efficiency.

References

[1] Wang Y, Lu W, Lou R, et al. Improving MapReduce Performance with Partial Speculative Execution [J]. Journal of Grid Computing, 2015, 13 (4): 587-604.

[2] White T. Hadoop: the Definitive Guide [M]. Southeast University Press, 2011.

[3] Hang L, Hang L, Hang L, et al. Neural generative question answering [C] // International Joint Conference on Artificial Intelligence. 2016.

[4] Hu-Sheng L, Shan-Shan H, Jun-Gang X U, et al. Survey on Performance Optimization Technologies for Spark [J]. Computer Science, 2018.

[5] Robert S, Patrick M, Phil T. Transparent fault tolerance for scalable functional computation [J]. Journal of Functional Programming, 2016:-.

[6] Wang K, Khan M M H. Performance Prediction for Apache Spark Platform [C] // IEEE International Symposium on IEEE International Conference on High Performance Computing & Communications. IEEE Computer Society, 2015.

[7] Kim M, Li J, Volos H, et al. Sparkle: Optimizing Spark for Large Memory Machines and Analytics [J]. 2017.

[8] Lu L, Xuanhua S, Yongluan Z, et al. Lifetime-based memory management for distributed data processing systems [J]. Proceedings of the VLDB Endowment, 2016, 9 (12): 936-947.

[9] Geng Y, Shi X, Pei C, et al. LCS: An Efficient Data Eviction Strategy for Spark [J]. International Journal of Parallel Programming, 2016.

[10] Chen K, Wang B, Feng L. Data object cache in Spark computing engine [J]. ZTE Technology Journal, 2016, 22 (2): 23-27.

[11] Shen B. Research on Spark caching strategy based on task structure optimization [D].

[12] Bian C. Research on significant technologies of performance optimization on in-memory computing frameworkD]. 2017.

[13] Bian C, Yu J, Ying C T, et al. Self-Adaptive Strategy for Cache Management in Spark [J]. Tien Tzu Hsueh Pao/Acta Electronica Sinica, 2017, 45 (2): 278-284.

[14] Hong-Tao M, Song-Ping Y U, Fang L, et al. Research on Memory Management and Cache Replacement Policies in Spark [J]. Computer Science, 2017.