Computers, Materials & Continua CMC, vol.61, no.2, pp.739-757, 2019

A Dynamic Memory Allocation Optimization Mechanism Based on Spark

Suzhen Wang[1], Shanshan Geng[1], Zhanfeng Zhang[1], Anshan Ye[2], Keming Chen[2], Zhaosheng Xu[2], Huimin Luo[2], Gangshan Wu[3, *], Lina Xu[4] and Ning Cao[5]

Abstract: Spark is a distributed data processing framework based on memory. Memory allocation is a focus question of Spark research. A good memory allocation scheme can effectively improve the efficiency of task execution and memory resource utilization of the Spark. Aiming at the memory allocation problem in the Spark2.x version, this paper optimizes the memory allocation strategy by analyzing the Spark memory model, the existing cache replacement algorithms and the memory allocation methods, which is on the basis of minimizing the storage area and allocating the execution area according to the demand. It mainly including two parts: cache replacement optimization and memory allocation optimization. Firstly, in the storage area, the cache replacement algorithm is optimized according to the characteristics of RDD Partition, which is combined with PCA dimension. In this section, the four features of RDD Partition are selected. When the RDD cache is replaced, only two most important features are selected by PCA dimension reduction method each time, thereby ensuring the generalization of the cache replacement strategy. Secondly, the memory allocation strategy of the execution area is optimized according to the memory requirement of Task and the memory space of storage area. In this paper, a series of experiments in Spark on Yarn mode are carried out to verify the effectiveness of the optimization algorithm and improve the cluster performance.

Keywords: Memory calculation, memory allocation optimization, cache replacement optimization.

1 Introduction

Spark is a memory-based distributed data processing framework. It is a fast and universal computing engine designed for large-scale data processing. During the execution of tasks on the Spark, it is necessary to load the data source to be processed from HDFS into RDD, and to generate the result by a series of RDD operations. Memory allocation is a key issue

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

[2] College of Mathematics and Computer Science, Xinyu University, Xinyu, 338004, China.

[3] School of Information Engineering, Jiangsu Polytechnic College of Agriculture and Forestry, Jurong, 212400, China.

[4] School of Computer Science, University College Dublin, Dublin 4, Ireland.

[5] College of Information Engineering, Sanming University, Sanming, 365004, China.

[*] Corresponding Author: Gangshan Wu. Email: gywgs@aliyun.com.

in memory computing framework. A good memory allocation scheme can effectively improve the resource utilization of memory and speed up the execution of tasks.

The existing memory management schemes include static memory partitioning and dynamic memory partitioning. Prior to Spark1.6, there was only static memory management and the size of the execution area and storage area cannot be changed. Memory allocation schemes are mainly FCFS and FA. In the FCFS algorithm, if one of the tasks has a large memory requirement, the other task cannot be allocated to memory, and the utilization of CPU is low. When the memory requirement of a task is very small, it will affect the utilization of memory, so the FCFS algorithm is suitable for the tasks of moderate and uniform memory requirement. The FA algorithm overcomes the disadvantages of low resource utilization and uneven memory allocation of FCFS algorithm to some extent, but it only considers the number of active tasks and does not consider the amount of data actually required by each task, causing the unfair in execution time of tasks, see Chen [Chen (2016)]. In recent years, many scholars have studied the memory allocation algorithm of Spark platform. For static memory management, an adaptive scheduling algorithm (SBSA) is proposed in Chen [Chen (2016)] based on FA algorithm, the times of tasks overflow and the overflow history of tasks. This study believes that tasks with more overflows requires more and will get more memory when there is free memory, but it does not take into account the initial memory allocation of task. Ying [Ying (2016)] also optimized the FA algorithm, and proposed an improved fair allocation algorithm, which is needed to distinguish the size of the task when tasks requests memory, for tasks with large memory requirements based on task overflow times and waiting time after task overflow. For dynamic memory management, Liao et al. [Liao, Huang and Bao (2018)] proposed that the storage space should be reduced without affecting RDD reuse and task execution efficiency, and the execution area should allocate memory according to the requirement of the task. Although this study improves the overall task execution efficiency to a certain extent, the speed of execution for tasks with small memory is reduced. Bian [Bian (2017)] points out that when the space allocated to the execution memory is larger, the job completion time is shorter.

Based on the above studies, the work done in this paper is as follows:

(1) Based on the goal of minimizing the storage area and allocating the execution area according to the demand, the weight calculation model of the RDD partition is improved, and the more valuable RDD partition is saved, thus the memory cache mechanism is optimized and the maximum memory is obtained in the execution area.

(2) In the execution area, according to the memory requirement of tasks, the task with small requirement of memory directly allocates memory, while the task with large memory requirement dynamically adjusts the whole memory allocation according to the memory demand and overflow times.

(3) The simulation results show that compared with the LRU algorithm, DWRP strategy and AWRP strategy, the optimized cache replacement mechanism is improved in three aspects: RDD computing cost, the times of cache replacement and memory utilization. By modifying and compiling the Spark source code, compiling and related experiments, the effectiveness of the optimized memory allocation algorithm is verified.

2 Correlation studies

2.1 Spark Memory allocation management

Spark Memory allocation is shown in Fig. 1. The default JVM heap size in Spark is 512M, which can be adjusted by the parameter *spark.executor.memory*. The memory here is basically shared by all tasks within the Executor, see Ying [Ying (2016)].

Spark Memory is the space needed to run the system framework, which is made up of two parts, Storage Memory and Execution Memory. After Spark1.6, Storage and Execution used Unified to use Spark Memory (Heap Size-300 MB) together. By default, Storage and Execution took up 50 percent of the space, and stored memory and executed memory shared the same space. You can dynamically adjust the respective memory areas. When there is not enough memory for execution, it can also spill some data to disk in addition to borrowing memory space from Storage Memory, see reference [Apache Spark (2018)]. The size of Execution Memory space directly affects the efficiency of job execution, so this paper mainly optimizes the RDD cache replacement algorithm based on Storage Memory to ensure the minimum space of Storage Memory and make the Execution Memory space maximum.

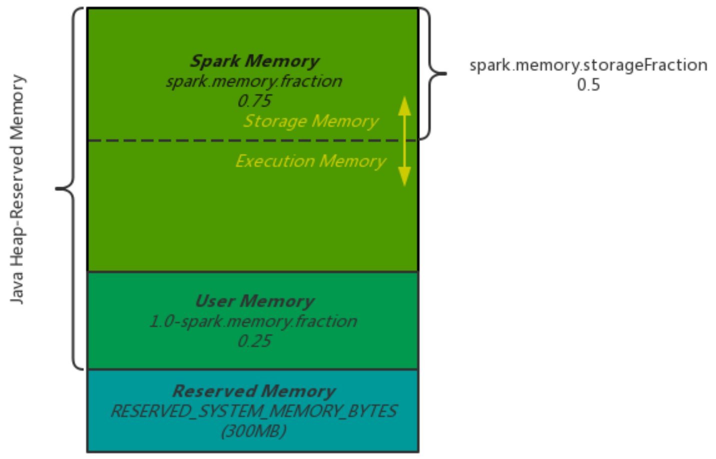

Figure 1: Spark memory allocation

2.2 Introduction to RDD

RDD (Resilient Distributed Data Set) is an immutable set of distributed objects. RDD can be logically abstracted into a HDFS file, but in fact, RDD can be divided into multiple RDD partitions and run on different nodes in the cluster. For developers, RDD can be seen as an object of Spark that itself runs in memory, such as reading a file is a RDD, evaluating files is a RDD and the result set is also a RDD. At the same time, RDD is fault-tolerant. If a RDD Partition on the node is lost because of the node failure, the RDD will recalculate the RDD Partition through its own data source. RDD is usually created from HDFS files or Hive tables, or from collections in an application.

Because the data in RDD may be TB, PB level, it is not realistic to store and compute on one node. Therefore, according to the distributed idea, Spark divides RDD into several subsets, that is, partition, which is similar to split in MapReduce. The number of partitions determines the granularity of the RDD calculation, and the calculation of partitions in RDD is performed in a single task.

RDD supports two types of operations, one is Transformation operations, and the other is Action operations, see Chen et al. [Chen, Li, Cao et al. (2016)]. Transformation operations (such as map operations, join operations, groupByKey operations) create new RDD for the next calculation. The Action operation, such as the reduce operations, count operation, is the last operation on the RDD. In Spark, when the RDD is converted, the operation is not executed immediately. The system records only the information about the conversion operation until the Action operation triggers the real operation. The RDD execution process is shown in Fig. 2.

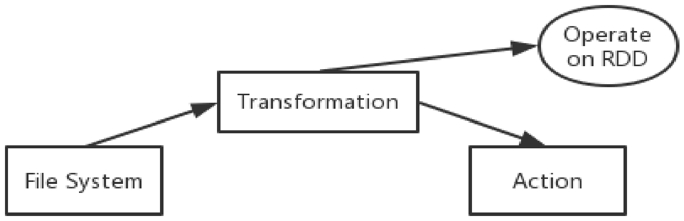

Figure 2: The RDD execution process

Because RDD is a coarse-grained set of operational data, each Transformation operation produces a new RDD, so that there are dependencies between RDD. There are two types of dependencies between RDD, Wide Dependency (or shuffle Dependency) and Narrow Dependency. Narrow Dependency means that one Partition of each parent RDD is used by up to one Partition of child RDD. Wide Dependency means that the Partition of a parent RDD is used by the Partition of multiple child RDD.

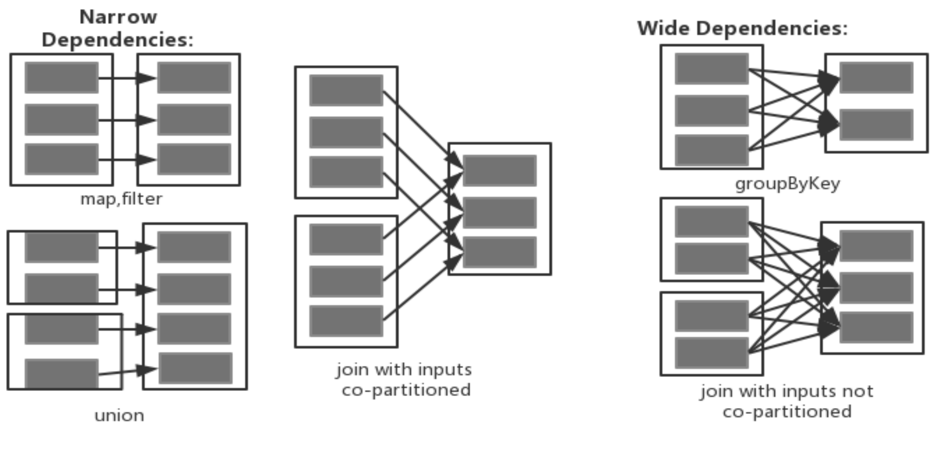
Narrow Dependencies:
Wide Dependencies:
map,filter
groupByKey
union
join with inputs co-partitioned
join with inputs not co-partitioned

Figure 3: Narrow dependency and wide dependencies

2.3 Research on Spark cache management

Since the memory space of the execution area directly affects the efficiency of the job execution, the memory occupied by the storage area should be minimized as much as possible. Spark uses RDD as the data structure, and the RDD is the abstraction of the memory data set by Spark, and the intermediate results produced in the course of Spark operation are stored in memory as RDD, see reference [Liao, Huang and Xu (2018)]. The cached data in the storage area is frequently defined by the user. When the data is no longer needed, it cannot be cleaned up in time, so it cannot effectively use the memory space. At the same time, when memory is fixed, more valuable RDD partitions should be cached to speed up the execution of tasks. Therefore, the storage memory should be reduced and utilized as much as possible to ensure that the execution area has sufficient memory resources to accelerate job execution efficiency. And if the RDD is persisted, then each node will leave the calculated RDD partition results in memory. If the RDD is used again in subsequent processes, it will not need to be recalculated, thus speeding up the execution of the action.

The persistence of RDD is handled by the Storage module of Spark, which realizes the decoupling of RDD and physical storage. The Storage module is responsible for managing the data generated by Spark in the process of calculation and encapsulates the functions of accessing data locally or remotely in memory or disk. In the implementation, the Storage module of the Driver side and the Executor side constitutes the architecture of master-slave, that is, the BlockManager on the Driver side is the Master and the BlockManager on the Executor side is the Slave.

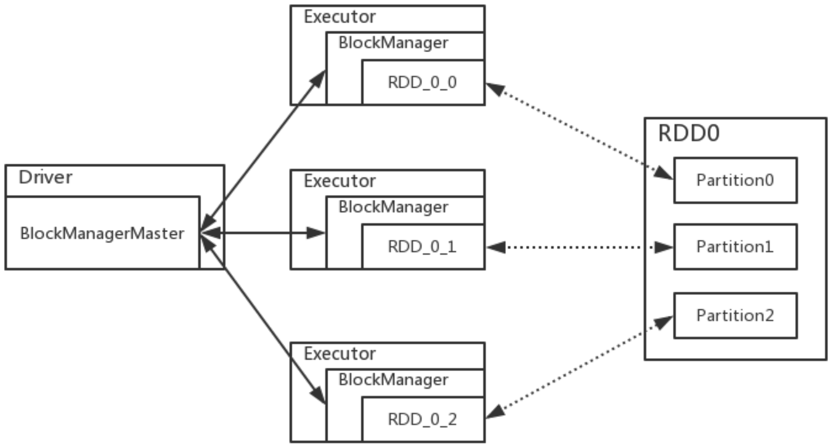

Figure 4: Storage module schematic

However, the use of memory in Spark is still in the primary and inefficient stages. The efficiency of memory usage depends entirely on the quality of the code written by the programmer. The programmer must specify the RDD to be cached. It is cached when the cache program statement is executed, see Meng et al. [Meng, Yu and Liu (2017)]. In the Spark framework, when the storage memory space is insufficient, the LRU algorithm is used by default to implement cache replacement, and the least recently used RDD is replaced.

In recent years, many scholars have studied the cache replacement algorithm for Spark platform. Liao et al. [Liao, Huang, Xu et al. (2018)] introduces RDD weight model.

Considering the RDD calculation cost and the the frequency of the RDD partition to be used, the Master records the RDD's initial calculation time and completion time by monitoring the RDD state change, that is, the RDD calculation cost. Liao et al. [Liao, Huang and Bao (2018)] considers the memory occupied by the RDD partition on the basis of that. Zhang et al. [Zhang, Chen, Zhang et al. (2017)] considers the distributed storage characteristics of the RDD partition. It is considered that when the Storage Memory space is insufficient, the complete RDD partition should be kept as much as possible. For the calculation of the RDD partition weight, the memory space occupied by the partition, access frequency and time characteristics are considered, but The RDD calculation cost is not considered. In Swain and Paikaray [Swain and Paikaray (2011)], according to the DAG diagram, the fine-grained RDD check-pointing and kick-out selection strategies are proposed to reduce the computational cost of RDD and maximize the memory usage during the selection of cache RDD. AWRP algorithm is designed and implemented by studying FIFO, LRU and other cache algorithms, see Duan et al. [Duan, Li, Tang et al. (2016)]. The algorithm calculates the weights of each cached object and selects the least recently used cache object. Meng et al. [Meng, Yu, Liu et al. (2017)] proposed the DWRP strategy considering the time when the RDD partition was stored in memory.

2.4 Spark execution memory allocation research

Spark 1.6 began to introduce dynamic memory management. The dynamic memory management mode is used in Spark by default. In the source code of *SparkEnv.scala*, if *spark.memory.useLegacyMode* is set to *true*, it will be changed to static memory management. There are two main scheduling modes in Spark: FIFO (first in, first out) and FAIR (fair scheduling). By default, the scheduling mode used in Spark is FIFO which means that the task submitted first is executed first, and the subsequent waiting tasks need to wait for the previous tasks to complete before continuing.

In the Spark work process, multiple jobs are generated for each Application submitted. A job can generate multiple stages through the demarcation point, and each stage contains multiple tasks. The memory allocation problem for tasks is similar to the job scheduling problem, which uses different scheduling methods to effectively utilize fixed resources.

3 Improved cache replacement algorithm

By minimizing the storage memory, the task execution efficiency can be effectively improved, and the RDDs that are no longer used are deleted from the cache list. If a new RDD is added to the cache list, and the size of the RDD stored in the cache list exceeds the cache memory, cache replacement is required.

Choosing a more valuable RDD is a core part of cache replacement, so sorting the RDDs in the cache list according to the principle of RDD’s importance is needed.

3.1 Influencing factors of RDD cache

Through the research of RDD cache replacement algorithm, the weights of RDD mainly have the following important factors:

(1) Total frequency of use of RDD: The number of times of the RDD appears in a DAG diagram.

(2) The frequency of RDD to be used: With the execution of the job, each time RDD is used, the frequency of RDD to be used minus 1. The higher the frequency of RDD to be used, the higher the value of being cached.

(3) RDD computing cost: The larger the computing cost, the higher the weight of RDD. The computing cost can be divided into the size of input data, the complexity of operation type and the complexity of closure (the concrete calculation content of RDD operation). The generation time $T_{RP_{ij}}$ for each partition is as follows:

$$T_{RP_{ij}} = ET_{ij} - ST_{ij} \tag{1}$$

where ET_{ij} represents the end time of each partition build and ST_{ij} represents the time that the partition start build. At the same time, the generation time of a RDD is determined by the longest generation time for all partitions, see Feng [Feng (2013)], so the computing cost of the RDD is：

$$T_{R_i} = \max\{T_{RP_{i1,}}..T_{RP_{ij.}}..T_{RP_{in}}\} \tag{2}$$

(4) The size of RDD partition capacity: Memory resource occupied by partition.

(5) Life cycle: The definition of life cycle in Xiong et al. [Xiong, Xia and Yang (2018)] is the number of Action experienced by a RDD from its inception to its destruction. In Liao et al. [Liao, Huang and Bao (2018)], it is defined as the time interval between the start and the end of each RDD. The maximum life cycle is from the start time of RDD to the time of job completion, and the minimum life cycle is the time from the start time of RDD to the last use of RDD.

$$\begin{aligned}
&\max LifeCycleRDDij = finishTimeRDDij - startTimeRDDij \\
&= finishjobi - generatedTimeRDDij \\
&\min LifeCycleRDDij = lastUsedTimeRDDij - generatedTimeRDDij \\
&= generatedTimelastChildofRDDij - generatedTimeRDDij
\end{aligned} \tag{3}$$

(6) The storage time: considering the idea of LRU algorithm, the longer a certain RDD partition is stored in memory, the more it is shown that it is a relatively “old” RDD partition. In other equal conditions, priority is given to the elimination of these “old” data, see Zhang et al. [Zhang, Chen, Zhang et al. (2017)].

3.2 Improved cache replacement algorithm

In the improved cache replacement algorithm, only four factors affecting RDD cache are selected: the frequency of RDD to be used, RDD computing cost, the size of RDD and the storage time. The specific algorithm process is as follows:

(1) When calculating weights, first convert partitioning and feature values to matrix form

Suppose there are *n* partitions in a storage memory, each partition has *m* characteristic attribute, in which four characteristics are given: partition to be used frequency *f*, partition computation cost *Cost*, partition size *S*, storage time *T*. The matrix is as follows:

$$RP = \begin{pmatrix} X_{11} & \cdots & X_{1m} \\ . & & \\ . & & \\ X_{n1} & \cdots & X_{nm} \end{pmatrix}_{n*m} \tag{4}$$

n represents the number of partitions, and *m* represents the characteristic attribute of each partition, here is 4. where X_{mn} represents the value of the *n-th* indicator of the *m-th* partition.

(2) Feature attribute normalization processing.Since the level difference between the four characteristic indicators is large, if the analysis is directly performed, the role of the higher value index in the comprehensive analysis will be highlighted, so that each index participates in the calculation with unequal weight. This paper uses the Max-min standardization method, see Wang et al. [Wang, Zhang, Zhang et al. (2018)]. The partition to be used frequency and partition cost calculation are positive correlation indicators, while the size and partition storage time are negative correlation indicators. Therefore, the indicators should be normalized and the conversion formula is as follows:

$$X'_{ij} = \begin{cases} \dfrac{X_{ij} - \min\{X_{ij},...X_{nj}\}}{\max\{X_{1j},...X_{nj}\} - \min\{X_{1j}..X_{nj}\}}, & \text{Positive indicators} \\ \dfrac{\max\{X_{ij},...X_{nj}\} - X_{ij}}{\max\{X_{1j},...X_{nj}\} - \min\{X_{1j}..X_{nj}\}}, & \text{Negative indicators} \end{cases} \tag{5}$$

(3) PCA dimensional reduction: by using the single feature and combined feature of RDD partition as its value, many experiments show that when one or more of these features are used, the experimental results are quite different. Because the cached RDD partitions in the storage space are constantly changing, the criteria that measure the value of the RDD partitions also need to be dynamically adjusted each time. In the cache replacement mechanism, only the two most important features of the partition in memory are retained at a time, so only the first two columns of principal components are required at a time.

The PCA algorithm is described as follows:

```
Input: sample set: $D = \{x_1, x_2, ...x_m\}$;
    Low-dimensional space dimension: d’
Process:
1. Centralization of all samples: $x_i \leftarrow x_i - \frac{1}{m}\sum_{i=1}^{m} x_i$;
2. Calculate the covariance matrix for the sample: $XX^T$;
3. Eigenvalue decomposition of covariance matrix;
4. Take the eigenvector corresponding to the maximum number of d’eigenvalues
Output: projection matrix: $W^* = (w_1, w_2, ...w_{d'})$
```

(4) Weight calculation equation: the product of the two strongest features of RDD partition.

$$\text{weight}i = wi1 * wi2 \tag{6}$$

(5) Finally, by calculating the weights of each partition, the partition with smaller elimination weight should be considered first when the partition is replaced. The main ideas are as follows:

① Get information about the RDD that needs to be cached.

② Put all RDD objects in the candidate replacement list and sort by weight, if the replacement list is empty, you do not need a replacement, and if not null, Step ③.

③ The RDD with the smallest option value in the replacement list begins to replace until the remaining RDD partition is smaller than the memory size and the computational cost of the replaced RDD is recorded.

4 Optimization strategy of execution memory allocation

Through the introduction of FIFO and FAIR scheduling algorithms, we can understand the characteristics of these two algorithms. During the execution of the tasks, if Execution memory is insufficient, in addition to choosing to borrow memory from the Storage Memory, a part of the data can be spilled to the disk, but the data is read from the disk much less than the data is read from the memory. Therefore, this paper optimizes the FAIR algorithm according to the memory demand of each task and the dynamic changes of the task during execution.

4.1 Correlation model

This section defines the following for Spark task execution and memory allocation:

(1) Job execution time. Spark is a memory-based distributed data processing framework. The main goal of using Spark is to improve job execution efficiency and reduce job execution time. Spark divides jobs into multiple stages, and each stage has one or more RDD, and each RDD also contains multiple partitions. The number of pipelines in each stage is determined by the number of partitions.

Suppose there are *n* stages, there are m RDD in *stagei*. Except for the last wide dependent RDDim, in each stage, the other RDD is divided into *x* pipeline, each pipeline is a task. In addition to the final wide dependent *RDDim* of each phase, the RDD is divided into *x* pipeline and each pipeline is a task. The collection of partitions for the pipeline is $\text{pipe}_{ix} = \{\text{pipe}_{ix1}, \text{pipe}_{ix2}, ..., \text{pipe}_{ixj}\}$, The execution time of each pipeline is $T_{\text{pipe}_{ix}} = \sum_{j=1}^{m-1} T_{p_{ixj}}$, the execution time of *stagei* is the sum of the maximum execution time of each pipeline and the computing time of *RDDim*: $T_{\text{stage}_i} = T_{\text{RDD}_{im}} + \max(T_{\text{pipe}_{i1}}, T_{\text{pipe}_{i2}}, ... T_{\text{pipe}_{ix}})$, the execution time of the job is $T_{\text{job}} = \sum_{i=1}^{n} T_{stage_i}$.

(2) Task parallelism. Task parallelism is the number of tasks executed concurrently at the same time. In the actual working environment, the degree of task parallelism is mainly determined by the number of working nodes and the number of CPU nodes. Assuming that the number of working nodes is *n* and the number of CPU cores per working node is

g, the maximum number of concurrency supported by the whole Spark environment is *n* * *g*, which is also called physical parallelism. If the user-specified parallel parameter is *L*, the minimum value priority should be followed, so the actual task parallelism can be expressed as:

$$\mathrm{dp}i = \min(L, n \times g) \tag{7}$$

When the user parallelism is greater than the physical parallelism, that is, the pipeline number of the stage is greater than the task parallelism degree, then the task needs to be assigned by multiple wheels, and the number of task assignment rounds can be expressed as:

$$R\mathrm{ound} = \mathrm{ceiling}\left(\frac{L}{\mathrm{n} \times g}\right) \tag{8}$$

(3) Amount of allocation memory in the execution area. The initial allocation of the execution area is related to the system setting parameters, which is determined by the parameter of *Spark.memory.fracton*. By default, *StorageMemory* and *ExecutorMemory* can be allocated the same amount 50%. Because the memory of the Storage and the memory of the Execution can be dynamically adjusted and the memory of the Storage can be borrowed from the execution area, the maximum amount of the allocated amount of memory in the execution area is the whole memory space. The memory allocation policy of the execution area directly affects the memory size that can be allocated by each task. In the FA algorithm, the memory size allocated by each task is the same. The sum of memory allocated for each parallel run of task cannot exceed the current amount of allocatable memory in the execution area.

$$\sum_{i=1}^{\mathrm{num}} Taski \le ExecutorMemory \tag{9}$$

(4) Amount of overflow memory in execution area. When the amount of memory allocated to *Taski* is less than the amount required for the task, the difference between them is the amount of memory overrun for the task.

$$\begin{gathered} \mathrm{sp}illMemoryTaski = TaskMemoryDemandi - executionMemoryAllocationTaski \\ spillMemoryExecutor = \sum_{i=1}^{n} TaskMemoryDemandi - executionMemoryAllocationTaski \end{gathered} \tag{10}$$

4.2 Detailed design of improved algorithm

In the improved memory allocation strategy, the weight is calculated according to the memory requirement and the number of overflows of the task, and then the weight is used to calculate how much memory should be allocated to the task.

Free memory refers to all memory in Execution memory that has not been used. The definition is as follows:

$$\mathrm{freeMemory} = \mathrm{maxPool} - \sum_{\mathrm{i=1}}^{\mathrm{m}} \mathrm{memoryForTask(i)}, \mathrm{m} \in \mathrm{M} \tag{11}$$

where *memoryForTask* stores all the unfinished tasks in the memory pool, *M* is the set of elements contained in *memoryForTask*, *i* is the id of the Task, and *MemoryForTask(id)* indicates the memory that the *Task* already occupies.

In the optimization algorithm, in order to ensure the execution efficiency of the task with small memory requirements, the minimum usable memory of each task is defined as *MemLow*, and the calculation equation is $Mem_{low} = \mathrm{maxPool}/(2*Task_{activity})$, where $\mathrm{maxPool}$ is the maximum memory of the execution area, and $Task_{activity}$ is the number of active tasks in the current Worker.If the memory size required by the task $Task_i$ is less than Mem_{low}, and the current free memory has enough memory allocated to the task, then the Executor only needs to allocate the amount of memory required for the task, namely:

$$Memgrant = Memapply(Memapply <= Memlow) \tag{12}$$

$$\mathrm{toGrant} = \min(\mathrm{Memgrant}, \mathrm{freeMemory}) \tag{13}$$

where Mem_{apply} refers to the minimum memory available of $Taski$, $\mathrm{freeMemory}$ refers to the free memory in the current Executor, and $\mathrm{toGrant}$ is the actual memory size obtained by $Task_i$.

For tasks whose memory requirements are greater than *MemLow*, the weight calculation is required. The share that a task can get from memory depends on the ratio of the number of task overflows to the total number of task overflows that have not completed running, and the ratio of the amount of memory required by the task to the sum of the amount of memory required by the current active task.

The weight calculation equation is as follows:

$$Weighti = \alpha 1 * (spillCounti / spillSum) + \alpha 2 * (Memapplyi / Memapplysum) \tag{14}$$

$$spillSum = \sum_{i=1}^{n} spillCount\mathrm{i} \tag{15}$$

$$Memapplysum = \sum_{i=1}^{n} Memapplyi \tag{16}$$

where $spillCount_i$ indicates the number of times the *i-th* task overflows, *spillSum* indicates the total number of task overflows that did not complete the run; $\mathrm{Mem}_{\mathrm{applyi}}$ indicates the amount of memory required for the *i-th* task, and *Memapplysum* indicates the sum of the amount of memory required for the currently active task.The maximum memory that a task can get depends on the number of overflows of the task and the amount of memory required, that is, the free memory weight of the task. The maximum memory that can theoretically be obtained is $Memhigh_i = \mathrm{maxPool} * \mathrm{Weight}_i$.The maximum memory that the current task can actually get is:

$$\max ToGrantk = \min(Memapplyk, \max(0, Memhighk - taskMemory(k))) \tag{17}$$

where $taskMemory(k)$ is the memory that the task has currently acquired.

Finally, if the current free memory has enough memory allocated for the task, then the Executor can allocate the maximum amount of memory needed for the task.

$$\mathrm{toGrant} = \min(\max ToGrantk, \mathrm{freeMemory}) \tag{18}$$

4.3 Execution flow of improved algorithm

In the improved algorithm, it is first determined whether the task exists in *taskMeomory*.

The *taskMemory* stores the task that has not been completed, the *taskMemory(id)* stores the memory occupied by the task, and the *spillCount(id)* stores the number of overflows of the task. If the current task has became an active task, it is needed to determine the memory required by the task. If the required memory is less than the minimum allocated memory guarantee, the maximum amount of memory allocated to the task is the amount of memory required for the task. If the required memory is greater than the minimum allocated memory guarantee, you need to calculate *maxPool*, *weight*, *MemHigh*, *maxToGrant* according to the relevant model definition in 4.2, and determine the *toGrant* to be returned.

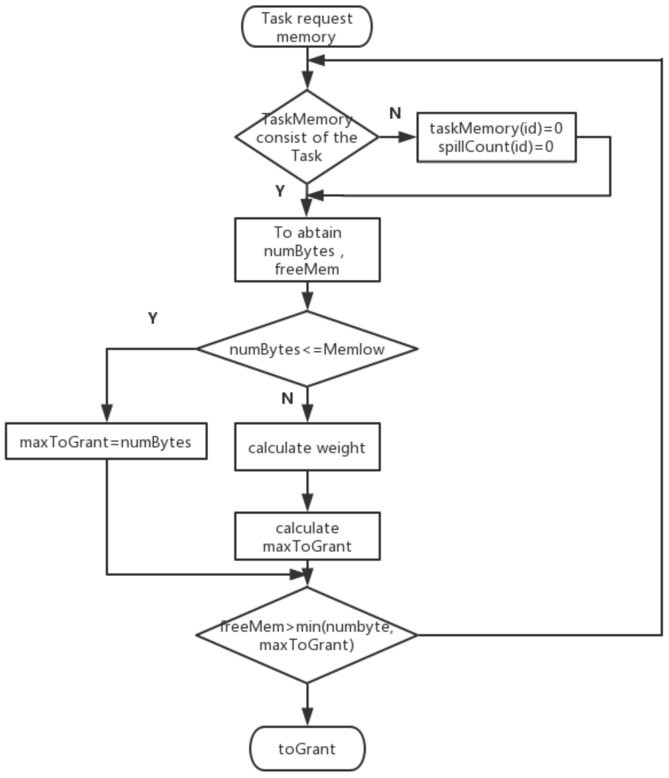

Figure 5: Improve memory allocation algorithm flow

The code is implemented as follows:

Improved Spark memory allocation algorithm

Input：

```
    Set of active tasks on Executor：activeTask
Algorithm：
    spillSum=0;
  weight=0.0;
  spill = new mutable.HashMap[Long, Int]()
  while (true) {
   val numActiveTasks = memoryForTask.keys.size
   val curMem = memoryForTask(taskAttemptId)
   maybeGrowPool(numBytes - memoryFree)
   val maxPoolSize = computeMaxPoolSize()
   if(spillSum!=0)
        weight=a*numBytes/maxPoolSize+b*spill(taskAttemptId)/spillSum
    val maxToGrant = math.min(numBytes, math.max(0, maxPoolSize *weight - curMem))
   else
   val maxToGrant = math.min(numBytes, math.max(0, maxPoolSize *numBytes/maxPoolSize - curMem))
   val toGrant = math.min(maxToGrant, memoryFree)
   if (toGrant < numBytes && curMem + toGrant < minMemoryPerTask) {
    logInfo(s"TID $taskAttemptId waiting for at least 1/2N of $poolName pool to be free")
    lock.wait()
   } else {
    Assign(memory);
    Refresh(taskMemoryInfo);
    Refresh(spill(taskAttemptId));
   }
  }
```

5 Experimental results and analysis

5.1 Improved cache replacement algorithm

This part is implemented by Matlab simulation. The dependencies between the various RDDs in Spark form a directed acyclic graph (DAG), so a DAG needs to be randomly generated. Considering that in different jobs, the characteristics of RDD size and generation cost are different. Therefore, each node (RDD partition) features are randomly generated, and Storage Memeory uses fixed size of memory space. In order to test and analyze the performance of the improved cache replacement algorithm, this paper implement the DWRP policy, AWRP strategy and LRU algorithm. In addition, in the experiment, we find that by PCA dimensionality reduction, it can be guaranteed that the improved cache replacement algorithm is less cost to calculate than each other algorithm in each experiment.

When the memory size allocated to the Storage Memeory is fixed, the number of cache replacements by using the DWRP policy, the AWRP policy, the LRU algorithm, and the improved cache replacement algorithm (NEWlru) is as shown in Fig. 6:

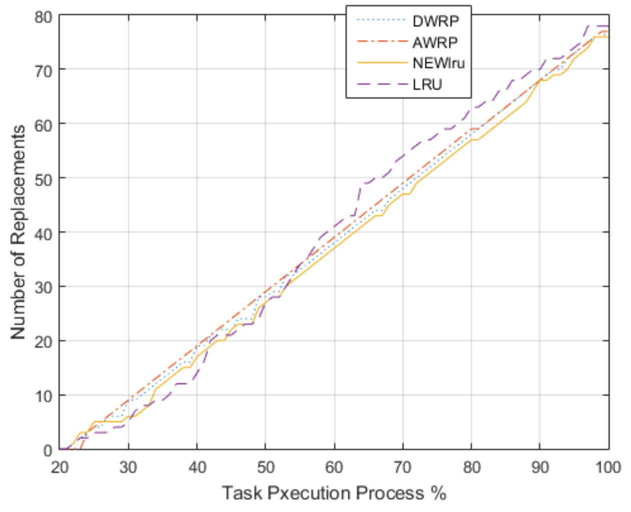

Figure 6: Number of replacements

When the memory is fixed, the improved cache replacement algorithm (NEWlru) has higher utilization of memory relative to the LRU algorithm, but the memory utilization is lower compared to the AWRP strategy as shown in Fig. 7.

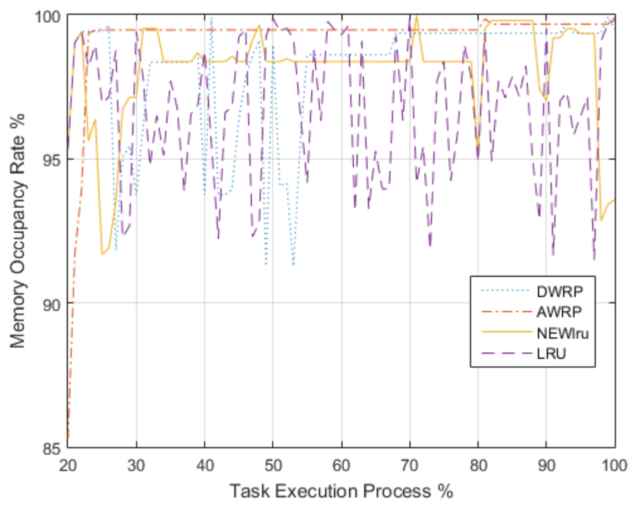

Figure 7: Memory occupancy rate

The computational cost of using the four algorithms is shown in Fig. 8:

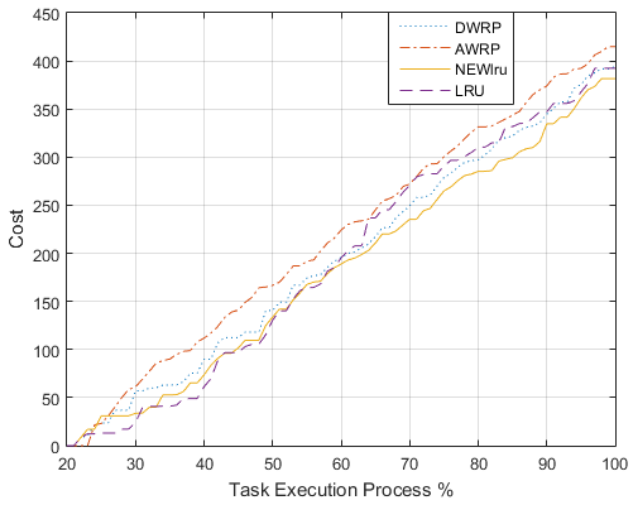

Figure 8: Cost

In addition, after many experiments, it is found that using the four features of the RDD partition, but not using PCA dimensionality reduction, does not guarantee the optimal results for each experiment.

In summary, the improved cache replacement algorithm not only can effectively improve memory utilization, reduce the number of cache replacements and the extra computational cost, but also ensure the cache effect in the case of RDD partition feature diversity.

5.2 Improved Spark memory allocation algorithm

In order to verify the effectiveness of the improved memory allocation algorithm, this study built Spark cluster and selected a variety of jobs for experiments. The experimental environment is shown in the following tables. Tab. 1 is the experimental cluster roles. Tab. 2 is the information of cluster. Tab. 3 is the information of the software version.

Table 1: Experimental cluster roles

IP	The name of the machine	Role
192.168.113.128	master	NameNode, secondaryNameNode, ResourceManager, Master
192.168.113.129	slave1	NodeManager, DataNode, Worker
192.168.113.130	slave2	NodeManager, DataNode, Worker

Table 2: Experimental cluster roles

The name of the machine	CPU cores	Memory (GB)
master	4	2
slave1	4	2
slave2	4	2

Table 3: Software version information

Software	Version
Centos	6.5
JDK	1.8
Hadoop	2.7.0
Spark	2.1.0

In the K-means experiment, the data set of Absenteeism at work was used. Three cluster centers were selected and the number of iterations was 100. The experimental results after multiple experiments on the three algorithms are shown in Tab. 4, where Best represents the minimum execution time in multiple experiments, and Mean represents the time average of multiple executions. It can be seen that the improved memory allocation algorithm (IM-FAIR) has similar task execution time compared to FIFO and FAIR algorithms when dealing with small data sets.

Table 4: Job execution time

JVM	FIFO		FAIR		IM-FAIR	
	Best	Mean	Best	Mean	Best	Mean
512 M	1.25	1.32	1.30	1.40	1.30	1.35
1024 M	1.20	1.22	1.20	1.26	1.20	1.26
1536 M	1.20	1.20	1.20	1.25	1.20	1.22

In the Wordcount experiment, the input data was 5 million lines and the file size was 547 M. In the three experimental environments of Driver memory size 512 M, 1024 M, 1536 M, the experimental results after multiple experiments on the three algorithms are shown in Fig. 9. It can be seen that when processing large data sets, as the JVM memory increases, the execution time becomes shorter and shorter, and the improved memory allocation algorithm (IM-FAIR) is superior to the FIFO and FAIR algorithms in terms of execution time.

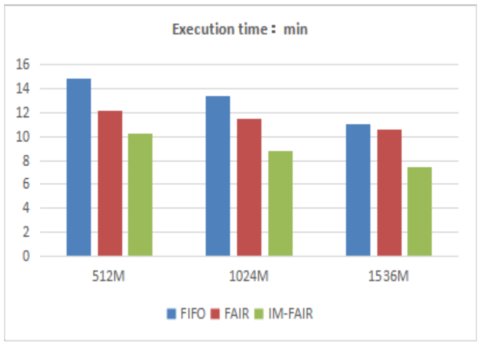

Figure 9: Execution time comparison

In the Wordcount experiment, the input data is 40 million lines and the file size is 4.62G. In the experimental environment with the driver memory size of 1536M and the task number of 37, the task completion time of the FAIR algorithm and the improved memory allocation algorithm is analyzed. The data processing time of the FAIR algorithm is 1.4 h, and the data processing time of the improved memory allocation algorithm (IM-FAIR) is 1.2 h. Fig. 10 is the Duration time of each task in the Stage0 stage, and Fig. 11 is the Duration time of each task in the Stage1 stage. It can be seen that in the improved algorithm, the average duration time of the task is less than the average duration time of the task in the FAIR algorithm, and the variation range of the average duration time is small.

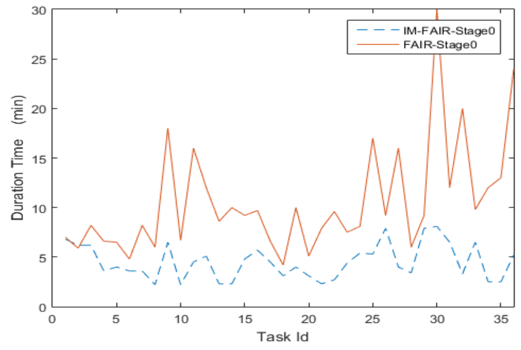

Figure 10: Task Duration time comparison in Stage 0

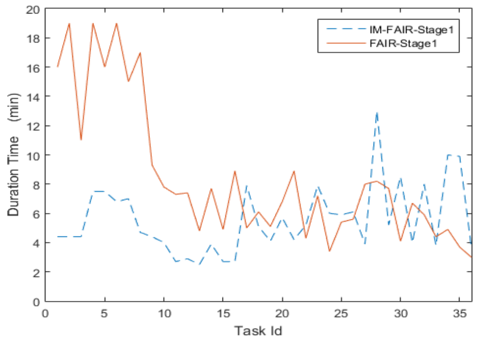

Figure 11: Task Duration time comparison in Stage 1

6 Conclusion

Based on the idea of storage area minimization and execution area on-demand allocation, this paper optimized the cache replacement algorithm according to the characteristics of the RDD partition and the PCA dimensionality reduction mechanism. At the same time, according to the memory needs of the Task, the memory allocation strategy of the execution area is optimized. Matlab simulation experiments show that the improved cache replacement algorithm is more effective than the LRU algorithm in terms of memory utilization, computation cost, and number of replacements. At the same time, the experiment is carried out on a Spark cluster, which verified that the memory allocation strategy proposed in this paper can effectively reduce the execution time of tasks and improve the execution efficiency. Next, we will verify the improved RDD partition cache replacement model in the Spark cluster, and further improve the job execution efficiency by combining the implemented memory allocation strategy.

Acknowledgement: This paper is partially supported by the Education technology Foundation of the Ministry of Education (No. 2017A01020), and the Major Project of the Hebei Province Education Department (No. 2017GJJG083), and the Science and Technology Project of Jiangxi Education Department (No. 2018GJJ181033). This work is also supported by the Jiangsu Polytechnic College of Agriculture and Forestry Key R&D Projects (2018kj11).

References

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

Bian, C. (2017): *Research on Key Technologies of Memory Computing Framework Performance Optimization (Ph.D. Thesis).* Xinjiang University, China.

Chen, Q. A.; Li, F.; Cao, Y.; Long, M. S. (2016): Spark task parameter optimization based on operational data analysis. *Computer Engineering & Science*, vol. 38, no. 1, pp. 11-19.

Chen, Y. Z. (2016): *Analysis and Optimization of Spark Shuffle Memory Scheduling Algorithm (Ph.D. Thesis).* Zhejiang University, China.

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.

Dabokele (2016): Spark scheduling mode-FIFO and FAIR.
https://blog.csdn.net/dabokele/article/details/51526048.

Feng, L. (2013): *Research and Implementation of Memory Optimization in Spark Cluster Computing Engine (Ph.D. Thesis).* Tsinghua University, China.

Hou, W. F.; Fan, W.; Zhang, Y. X. (2017): Improved Spark Shuffle memory allocation algorithm. *Journal of Computer Applications*, vol. 37, no. 12, pp. 3401-3405, 3429.

Liao, H. S.; Huang, S. S.; Xv, J. G.; Liu, R. F. (2018): A survey of Spark performance optimization technology. *Computer Science*, vol. 45, no. 7, pp. 7-15, 37.

Liao, W. J.; Huang, Y. F.; Bao, C. K. (2018): Memory optimization of Spark parallel computing framework. *Computer Engineering & Science*, vol. 40, no. 4, pp. 587-593.

Lin, Z. Y.; Lai, M. X.; Zou, Q.; Xue, Y. S.; Yang, S. Y. (2013): Flash database buffer replacement algorithm based on replacement probability. *Chinese Journal of Computers*, vol. 36, no. 8, pp. 1568-1581.

Meng, H. T.; Yu, S. P.; Liu, F.; Xiao, N. (2017): Spark memory management and caching strategy research. *Computer Science*, vol. 44, no. 6, pp. 31-35, 74.

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.

Pan, F. F.; Xiong, J. (2018): NV-Shuffle: Shuffle mechanism based on non-volatile memory. *Journal of Computer Research and Development*, vol. 55, no. 2, pp. 229-245.

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

Wang, S. Z.; Zhang, Y. P.; Zhang, L.; Cao, N.; Pang, C. Y. (2018): An Improved Memory Cache Management Study Based on Spark. *Computers, Materials & Continua*, vol. 56, no. 3, pp. 415-431.

Xiong, A. P.; Xia, Y. C.; Yang, F. F. (2018): A shuffle optimization mechanism under Spark cluster. *Computer Engineering and Applications*, vol. 54, no. 4, pp. 72-76.

Ying, C. T. (2017): *Research on Storage Layer Fault Tolerance and Optimization Strategy in Memory Computing Environment (Ph.D. Thesis).* Xinjiang University, China.

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.

Zhang, X. W.; Li, Z. H.; Liu, G. S.; Xu, J. J.; Xie, T. K. (2018): A Spark Scheduling Strategy for Heterogeneous Cluster. *Computers, Materials & Continua*, vol. 55, no. 3, pp. 405-417.