ATLANTIS PRESS

Advances in Engineering, volume 126

5th International Conference on Machinery, Materials and Computing Technology (ICMMCT 2017)

Research on Serialization Storage Strategy Based on Spark Cluster

Yang Fangfang[1,a], Xia Yuchong[1,b]

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

[a]413621484@qq.com,[b]357917886@qq.com

Keywords: Spark; Memory; Operator; RDD; Caching

Abstract. Spark is a kind of big data processing platform based on memory computing. The Spark default serialization strategy has low utilization of cache which has greatly influenced the efficiency of Spark task execution. For solving this problem of low computational efficiency caused by insufficient memory, this paper proposes an optimized serialized storage strategy, which combining with the running cot of RDD, RDD execution time and count of Action. Experimental results show that the proposed strategy can improve the computational efficiency under the limited task resources.

Introduction

With the era of big data coming, the ecosystem of big data processing platform constantly is constantly updated [1]. As MapReduce [2] can only support Map and Reduce operations, which has low efficiency of iterative computation, and limitations during the iterative processing and stream computing. Thus, an efficient distributed computing framework, Apache Spark [3], which can be used for batch processing, stream computing and interactive computing, emerges as the times require. This framework uses Resilient Distributed Datasets (RDD [4]) based on the cache for iterative computation to improve computational efficiency.

Currently, Spark has been widely deployed in many domestic enterprises. For instance, Tencent used Spark to calculate the number of shared friends between two QQ users in order to predict the user's ad click probability and improve the efficiency of the entire structure by ETL (Extract-Transform-Load) SparkSQL and DAG tasks; China Telecom introduced the Spark to ensure the efficiency of query while allowing the system to have parallel expansion capability in the 360-degree feature extraction of customers; and YouKu greatly shortened time required for task submission calculations by using Spark / Shark in BI and Ad serving. However, due to the limitation of cache resource, some parts of data need to be serialized to the disk while processing large data sets, which would deeply affect the computing performance. Therefore, a reasonable and efficient serialization strategy become an urgent problem in order to improve the iterative computation efficiency.

Related Work

Most Spark programs have the ability of memory calculating, so all the resources in the cluster: CPU, network bandwidth and memory are likely to become bottlenecks in the Spark program [5]. In the iterative computation, it is best to load all the data into memory to improve the computational efficiency, but in the big data computing environment, it surely exists large dataset and the lack of limited cache resources, so the dataset serialized storage becomes the key [6].

Feng Lin et al [7] proposed to decide whether to keep the RDD in cache according to the number of RDD usage, but the algorithm is not accurate enough to calculate the RDD weight, and the efficiency of distributed environment is not obvious; The graph algorithm proposed by Li Wenzhong et al. [8] is used to optimize the cache in distributed cache systems. However, this algorithm is only for the problem of cache placement, not for how to select data efficiently. Chen Yingzhi [9] proposed Spark shuffle memory scheduling algorithm, designed calculation of idle memory in detail: The key task can be borrowed from free memory and the maximum threshold of

task memory available, but this algorithm only optimized parameters based on fair scheduling algorithm, which cannot achieve an optimal efficiency.

Most of the existing researches focus on Spark placement strategy and optimization of scheduling algorithm, so as to be able to keep high efficiency in the case of insufficient memory. This paper combines the data characteristics of the computing framework in the implementation process, running cost, RDD execution time and the number of crossing Action, etc., to study the cache dataset serialized storage strategy in order to enhance the efficiency of calculation task.

Serialized Storage Strategy

Problem Analysis.

Since data in the Spark is transformed and processed in the form of RDD, Spark's dependency is divided into two types: wide dependency and narrow dependency [10]. For wide dependency, the application default serializes all the required RDD datasets into memory. Some operators, such as join, union and reduceByKey, put the entire dataset in memory for processing, so the application is prone to lack memory resources. And for narrow dependency, though less memory in dataset transformation, if we give priority to cache the data which uses many times, when the memory is limited, the remaining resources can serialize storage, so that the efficiency of entire application can be directly improved. Therefore, in the process of selecting serialized objects, the operator becomes an important factor.

In order to improve the cache utilization, it is necessary to ensure that RDD objects are less involved during the RDD serialization process, and keep the RDDs that need to be iterated or used more than once in the cache. However, In the process of task scheduling, we need to use different operators according to different business logic to deal with the RDD dataset, and because of different internal implementation of different operators, the efficiency of RDD dataset transformation is different, and size of dataset is also different, in addition, RDD's life cycle (the number of crossing Action) also plays a key role in the choice of serialized storage strategy. Therefore, the choice of RDD serialization will be affected by factors such as running cost of the operator, execution time of RDD, and the number of RDDs crossing the Action.

RDD Serialization Storage Strategy.

According to the analysis above, we choose size of RDD $Size(RDD)$, calculation cost of RDD, execution efficiency of RDD and the number of action AN as the impact factors to build the model of serialization storage strategy. We give the method of calculation on RDD operator cost, RDD execution efficiency and the number of RDD across Action.

(1) influence factor of operator

Since it is difficult to obtain the computational cost of operator, we use the analytic hierarchy process [11] to analyze the relationship between time complexity and spatial complexity of the operator. Here, we divide into four categories based on whether the operator can trigger the shuffle operation: one-to-one, one-to-many, many to many and whether to trigger the calculation process. The specific classification and relative weights are shown in Table 1.

Table 1 Cost Analysis of Common Operator Calculation

Operation Type	Category	Operator	Relative Weight
transformation	one-to-one	map(f:T=>U)	a1
		flatMap(f:T=>Seq[U])	a2
	one-to-many	groupByKey()	a3
		reduceByKey()	a4
	many-to-many	union()	a5
		join()	a6
action	trigger calculation	count()	No
		collect()	
		save(path:String)	

In the process of calculating complexity factor, we first establish the hierarchical model. By using arithmetic mean, we can get the complexity factor when calculating the operating cost.

(2) RDD implementation efficiency

In this paper, RDD $= \{RDD_1, RDD_2, \ldots, RDD_n\}$ represents the set of RDD, when building the spark cluster, we deploy several virtual machines, and all of them are of the same type. So the processing capacity of each virtual machine is the same. Because memory is variable, the processing capacity changes with different memory, and we use P_{mem} to denote the processing capacity. The execution time is estimated using the size of the j-th partition of RDD_i, which is called S_{ij}. Assuming that each RDD has m partitions, $\{P_{i1}, P_{i2}, \ldots, P_{im}\}$ are the partitions of RDD_i , and we can approximate partition execution time $ET_{P_{ij}} = \frac{S_{ij}}{P_{mem}}$.

Since all partitions in each RDD are executed in parallel, the execution time of RDD is the time that the longest partition is executed, that is,

$$ET_{RDD_i} = \max\{ET_{P_{i1}}, ET_{P_{i2}}, \ldots, ET_{P_{im}}\} \quad (1)$$

Where *m* indicates that there is *m* partitions in the i-th RDD in total.

As the application runs in the cluster, some partitions need to pull data from other nodes, so we need to consider the communication time between nodes, where we use the subtraction between completion time FT_{ij} and start time ST_{ij}, that is: $FT_{ij} - ST_{ij}$, and N_{ij} represents the number of RDD partition, so we can get the implementation efficiency of entire RDD:

$$E_{RDD_i} = \sum_{j=1}^{m} E_{P_{ij}} = \sum_{j=1}^{m} \frac{(FT_{ij} - ST_{ij}) * N_{ij}}{S_{ij}} \quad (2)$$

(3) RDD number of crossing Action *AN*

Since all transformation in Spark is inert which does not directly calculate the results. On the contrary, they just remember to apply these transformation to the underlying dataset (for example, a file). These transformation will only run when a response is required to return the result to the Driver. Because only when RDD encounter Action operator, the application will trigger the calculation, so we can judge according to each RDD across the Action. As calculating *AN*, the Action and each Action before the RDD transformation will be stored in the Map, so in the forward and backward cycle by the Action where the location is recorded as *i*, from the back before the loop to get the action *j*, so *j-i* is the relative number of RDD across the Action.

Definition 1 (operator weight) Define W_i(i=1,2,…,M) to represent the operator weight, where M represents the number of operators. Each operator has a weight, and according to the above analytic hierarchy process, the metric relation $C_v = f(O_t, O_s)$ between the time complexity and spatial complexity of the operator can be obtained. The larger the value, the greater the cost of the operator calculation. Accordingly, the probability of selecting serialized storage is smaller.

$$W_i = \frac{C_v * O_{t_i} + O_{s_i}}{(C_v * O_{t_i} * O_{s_i}) + 1} \quad (3)$$

Where, O_t represents time complexity of operator, O_s represents spatial complexity of operator, and C_v represents the measure of time complexity and spatial complexity.

Definition 2 (RDD weight) Define $W_{RDD_{i(a)}}$ to represent weight of the i-th RDD dataset after the operator *a*. If RDD is not in memory, it needs to be calculated in real time when used, and the calculation cost of different RDD is different, according to the definition of this paper, the greater the calculation cost, the greater the weight, so the RDD should have more opportunities to stay in the cache, on the contrary, you need to serialize to disk.

$$W_{RDD_{i(a)}} = k * \frac{W_i * AN * E_{RDD_i}}{Size(RDD)} \quad (4)$$

Where, $Size(RDD)$ denotes the size of the RDD, W_i denotes the weight of the i-th operator, *AN* denotes the number of actions passed by the operator, E_{RDD_i} represents the processing time of the i-th RDD, and k denotes the calibration parameter which values {10,100,1000, ...}.

According to the modeling method we mentioned above, we can analyze the influence of the operator on the whole dataset, so as to determine the weight of the operator and RDD, and thus decide whether to cache according to the size of the weight. What we consider is the lack of memory resources, so the choice of data is strict. The current memory cannot meet the calculation,

so the implementation of the strategy is needed to consider whether the current memory reaches a certain threshold when the strategy is triggered. The storage strategy of this paper shows as follows:

1) Use ganglia to detect the memory usage of machine during the execution of application. If the current memory is detected to be normal, the monitoring will continue. If the specified threshold is detected, the custom serialization storage strategy is triggered;
2) The size of the data set is $Size(RDD)$. According to the formula (2) and (3), we can get the corresponding parameter E_{RDD_i}, W_i;
3) According to the formula (4), we get the sorted RDD sequence $\{sorted(W_{RDD_i})\}$, that is, serialized candidate set;
4) Choose the smallest value of serialized storage from the serialized candidate set, that is, $min\{sorted(W_{RDD_i})\}$;
5) Continue step 1) until the task is completed.

Since we need to find the RDD cache which calculates more and the smaller size, all the algorithms need to be singled out, that is, do not use chain programming principles, so that RDD can be generated to calculate for every step. In the calculation process, the cost of operator is greater, the number of Action is more, the RDD execution rate is bigger, the size is smaller, and we need to cache it seriously. On the contrary, the RDDs need to be serialized to the disk.

Experiment and Result Analysis

We use two different sets of cluster to run different tasks. First, the cluster uses six nodes, the configuration is as follows: 16 core central processing unit (CPU), 12G memory, 500G hard drive, network bandwidth 1000M, 64-bit operating system CentOS 6.0. PageRank is used to verify.

Runtime Comparison.

In the experiment, we select 1G, 3G, 6G, 10G, 15G datasets. According to the above analysis of RDD weights, the average runtime of normal application in comparison with using the serialization storage strategy one, which can be obtained the results of Figure 1 as shown.

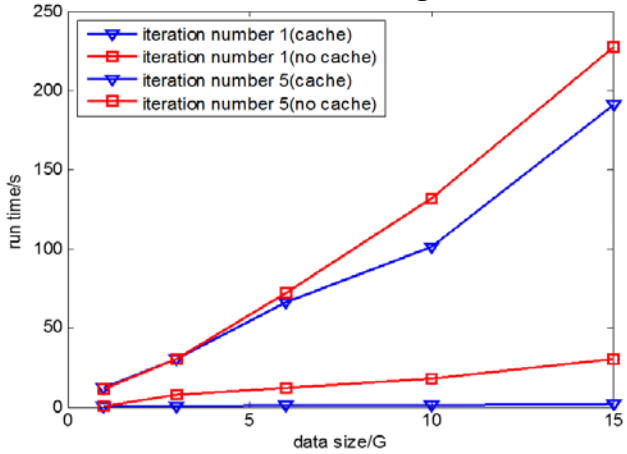

Fig. 1 Experiment Result of PageRank

In Fig 1, we can see that a polyline with a square line represents the result of no serialized storage, and a triangular polyline indicates the result of adding the serialized storage strategy. When iteration time is 1, it is necessary to read the data source from hdfs. Therefore, there is little difference between the two polylines at the top and bottom, that is, when using serialized storage and without serialization, the operation has little effect on the running result. When iteration time is increased to 5, the running time is greatly reduced since the intermediate variable data can be taken from the memory, which can get a speed ratio of about 15 times.

The experimental results show that in the iterative calculation of application, selecting the data which still uses in the next iteration process can enhance the execution time of entire application.

Memory Usage Comparison.

We still use the PageRank algorithm to test the system performance under different conditions. We use four group data which is G1 (130000, 840000), G2 (82000, 950000), G3 (260000, 1200000),

and G4 (280000, 2300000) to record the memory usage of the system separately and observe memory usage under stable conditions with ganglia, and the results are as follows.

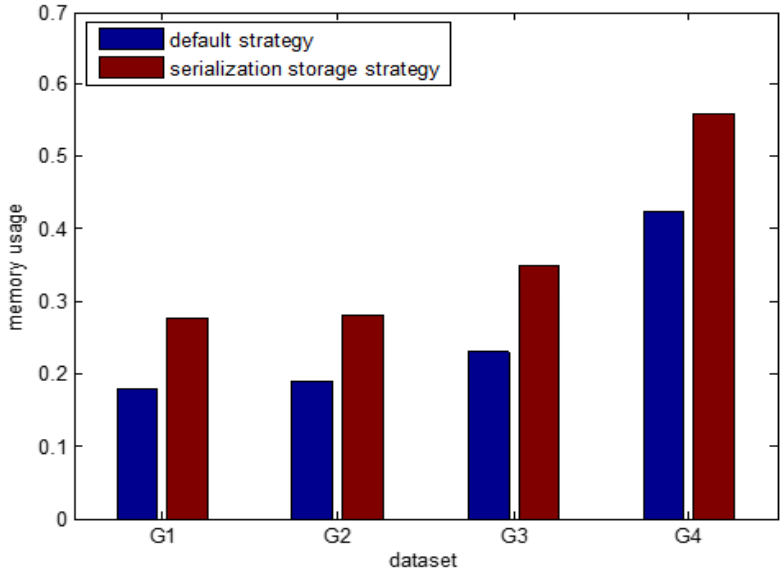

Fig. 2 Comparison of Memory Usage

In Fig 2, we can see that the memory usage of the application using serialized storage strategy is significantly higher than that of unused serialized storage strategy. According to our serialized storage strategy, we can put RDDs into cache which use many times, and when the RDD reappears again, we do not need to recalculate them, which reduces the system overhead and accelerates the run time of entire iteration. When adding nodes, there is little memory to store intermediate data until the last iteration ends. And PageRank is data-intensive application, it needs more memory resources.

The experimental results show that default Spark serialization storage strategy is very random and cannot make full advantage of memory, and the improved algorithm can improve the memory utilization of application and thus improve the execution efficiency of the application.

Conclusion

Spark is an efficient distributed computing framework, especially for data-intensive applications, because of its memory-based computing, iterative computation has more advantages. However, while selecting the appropriate RDD for serialized storage, the memory resources are not fully utilized. Therefore, serialized storage strategy in this paper can improve the computational efficiency in data-intensive computing. Future work and research will be focused on the serialization storage of multi-node in the cluster and the memory utilization problem combining with the task locality, in order to further improve the entire cluster computational efficiency.

References

[1] Chang B R, Tsai H F, Wang Y A. Optimized Multiple Platforms for Big Data Analysis[C]// IEEE Second International Conference on Multimedia Big Data. IEEE Computer Society, 2016:155-158.

[2] Yang Zhiwei, Zheng Quan, Wang Song, et al. Adaptive Task Scheduling Strategy for Heterogeneous Spark Cluster[J]. Computer Engineering, 2016, 42(1):31-35.

[3] Zaharia M, Chowdhury M, Franklin M J, et al. Spark: cluster computing with working sets[C]// Usenix Conference on Hot Topics in Cloud Computing. USENIX Association, 2010:10-10.

[4] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing[C]// Usenix Conference on Networked Systems Design and Implementation. USENIX Association, 2012:141-146.

[5] Islam N S, Lu X, Wasi-Ur-Rahman M, et al. Can Parallel Replication Benefit Hadoop Distributed File System for High Performance Interconnects?[C]// High-Performance Interconnects. IEEE, 2013:75-78.

[6] Duan M, Li K, Tang Z, et al. Selection and replacement algorithms for memory performance improvement in Spark[J]. Concurrency & Computation Practice & Experience, 2015, 28(8):2473-2486.

[7] Chen Kang, Wang Bin, Feng Lin. Data Object Cache in Spark Computer Engine[J]. ZTE TECHNOLOGY JOURNAL, 2016, 22(2):23-27.

[8] Li WZ, Chen DX, Lu SL. Graph-Based optimal cache deployment algorithm for distributed caching systems. Journal of Software, 2010, 21(7):1524-1535.

[9] Chen Yingzhi. Analysis and Optimization of Memory Scheduling Algorithm of Spark Shuffle[D]. Zhejiang University, 2016.

[10] Rana N, Deshmukh S. Shuffle Performance in Apache Spark[C]// International Journal of Engineering Research and Technology. ESRSA Publications, 2015.

[11] Deng Xue, Li Jiaming, Zeng Haojian, et al. Research on Computation Methods of AHP Weight Vector and Its Applications[J]. MATHEMATICS IN PRACTICE AND THEORY, 2012, 42(7):93-100.