CSAS: Cost-Based Storage Auto-Selection, a Fine Grained Storage Selection Mechanism for Spark

Bo Wang, Jie Tang, Rui Zhang, Zhimin Gu

CSAS: Cost-based Storage Auto-Selection, A Fine Grained Storage Selection Mechanism for Spark

Bo Wang[1], Jie Tang[2], Rui Zhang[1,3], and Zhimin Gu[1]

[1] Beijing Institute of Technology University, Beijing 100081, P.R.China
[2] South China University of Technology University, Guangzhou 510641, P.R.China
cstangjie@scut.edu.cn
[3] Yan'an University, Yan'an 716000, P.R.China

Abstract. To improve system performance, Spark places the RDDs into memory for further access through the caching mechanism. And it provides a variety of storage levels to put cache RDDs. However, the RDD-grained manual storage level selection mechanism can not adjust depending on computing resources of the node. In this paper, we firstly present a fine-grained automatic storage level selection mechanism. And then we provide a storage level for a partition based on a cost model which fully considering the system resources status, compression and serialization costs. Experiments show that our approach can offer a up to 77% performance improvement compared to the default storage level scheme provided by Spark.

Keywords: big data, spark, storage level selection, optimize

1 Introduction

To balance volume and speed, Spark [1] provides five flags to mark storage level, corresponding to whether use disk, memory, offHeap, serialization and replication. However, the storage level selection mechanism of Spark has the following two problems: Firstly, storage level of a RDD is set by programmers manually, by default storage level is MEMORY_ONLY which the RDD can only be cached in memory. Experiments show that there are significant performance differences among different storage level. A reasonable storage level decision results in performance improvements; A wrong decision can lead to performance degradation or even failure inversely. Secondly, RDD-grained storage level selection mechanism may lead lower resource utilization. In Spark, the same cached RDD uses the same storage level, on the contrary, different RDDs may use different storage levels. While a RDD is divided into several partitions which have different size and locate in different executors. Some partitions of a RDD may be computed on executors which have enough free memory, and others will be pended on executors which have not enough free memory on contrary.

In this paper, we propose a fine-grained storage level selection mechanism. Storage level is assigned to a RDD partition, not RDD, before it cache. And

storage level selection of a RDD partition is automatically basing on a cost model which takes fully account of memory of the executor and various computing costs of the partition.

2 Design and Implementation

2.1 Overall Architecture

CSAS (Cost-based Storage Auto-Selection) can wisely select a Storage-level, based on future costs, for a partition before it is to be cached. The overall architecture, shown as Fig.1, consists of three components: *(i)Analyzer*, which lies in the driver, provides the function of analyzing the DAG structure of the application to obtain the RDDs which will be cached and their execution flows; *(ii)Collections*, one in each executor, are used to collect real-time information, such as creation time, (De)serialization time of each RDD partition, during the task running; (*iii) Storagelevel Selectors*, also one in each executor, are arbiters for decision which storage level will be used by RDD partition when they will be cached.

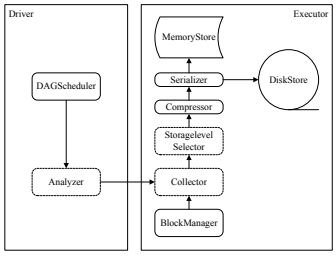

Fig. 1. Overall architecture of CSAS.

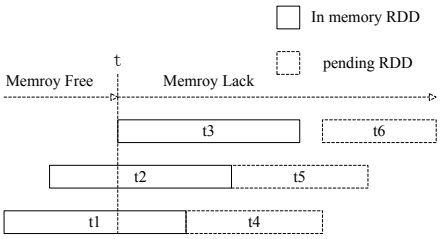

Fig. 2. RDD execution model.

2.2 Analyzer

As mentioned above, *Analyzer* obtains dependencies of RDDs and RDDs that will be cached according to DAG constructed by DAG Scheduler. There are two types of RDDs, called cache RDDs and non-cache RDDs. When computes on a RDD, it is necessary that all RDDs it depends are ready. If there are RDD(s) absent, the absences need to be created firstly. For non-cache RDDs, they are computed each time; For cache RDDs, they are got from memory or disk or both through CacheManager according of storage level after first computing. So we need to obtain the interdependencies among cache RDDs. In this paper, we use LCS's DFS algorithm [3] to get the ancestor cache RDDs of a RDD and the creating path of each cache RDD. At last, all these informations are recorded and are used to compute create cost of a RDD in *Collector*.

2.3 Collector

Informations that A *collector* collect are listed in the followings:

Create Cost: Time spends on computing a RDD partition after all ancestor cache partitions are ready, denoted as C_{create}.
(De)Serialization cost: Time takes on serialize or deserialize a RDD partition, denoted as C_{ser} or C_{deser}.
(De)Compression cost: Time takes on compress or decompress a RDD partition, denoted as C_{comp} or C_{decomp}.
Disk cost: Time takes on I/O on Disk, denoted as C_{disk}.

When (De)Serialization cost of a RDD partition is absent, we need estimate it using its cost per MB data [3], denoted as SPM and DSPM. When (de)serialization cost is unknown for a RDD partition, we estimate its (de)serialization cost according by the size of the partition and SPM or DSPM of corresponding RDD respectively. (De)Compression cost is also estimated using the same way.

According the above definition, we can calculate cache cost of a RDD partition in different scenarios, denoted as C_{cache}. To get optimum storage level for each partition, we compute cache cost of a RDD partition of *Normal scenario*, *Serialization scenario*, *Disk scenario* and *Compression scenario* to determine each storage level flag of a partition.

As shown in Fig.2, RDD partitions in memory will reach saturation at the time t. At this time, the pending RDD partitions should wait until some tasks finished and freed enough space to run. So the whole computing is divided into parallel computing and sequence computing two phases in a stage. Among them, computing operations on RDD partitions have no interference each other in parallel computing phase. In contrast, operations on a RDD partition must delay until another computing finished in sequence computing phase. The moment that sequence computing phase begins is after the first finish task in parallel computing phase has released its memory. Thus, the worst case caching cost in stage i can be concluded in 1:

$$C^i_{wccc} = max\{C^1_{cache}, ..., C^m_{cache}\} + \sum_{(k=1)}^{(n-m)/m} maxC^k_{cache} \tag{1}$$

Where n is the number of total RDD partitions will be cached in the future of this stage; m is the number of RDD partitions computing in parallel; $maxC^k_{cache}$ is one of the top $(n-m)/m$ max cache cost among RDD partitions in sequence computing phase. The second half is the sum of top $(n-m)/m$ cache cost among RDD partitions in sequence computing phase.

2.4 Storagelevel Selectors

Storage level selector in this executor evaluates an appropriate storage level for the partition based on worst case caching cost of various scenarios before a RDD partition is to be cached. Algorithm 1 shows the storage level selection strategy

for a cache RDD partition is determined by the values of worst case caching cost among various scenarios under the current memory circumstance. All costs used in algorithm 1 are caculated based on 1.

Algorithm 1. Cost-based Storage Level Selection

```
function StorageLevelSelection
Input: Cms_regular, Cms_serialize, Cms_compress, Cms_disk,
 unroll_size, free_size
Output: StorageLevel
  useMemory = true
  deserialized, useDisk, Compress = false
  if(Cms_regular > Cms_serialize)then
    deserialized = true
  endif
  if(Cms_regular > Cms_disk)then
    useDisk = true
  endif
  if((Cms_compress < Cms_serialize)
&&(Cms_compress < Cms_disk < 0))then
      Compress = true
  endif
  if(unroll_size > free_size)then
    useMemory = false
  endif
  if(!useMemory)then
    useDisk = true
  endif
  return StorageLevel
endfunction
```

3 Performance Evaluations

The experiment platform includes a cluster with three different nodes, one as both master and executor and the remaining only act as executors. And we adopt HDFS for storage, each partition has one replications. The datasets are generated by BigDataBench [4]. We use WordCount and KMeans two benchmarks in our experiments. For the convenience of test, we have two RDD cache for wordcount, respectively textFileRDD and flatMapRDD. The size of the two RDDs is about nine times difference. All data are normalized based on CSAS execution time.

Fig. 3 shows the difference of performance between CSAS and Spark native system which under different cache storage levels. It shows that there is a huge difference in execution time under different storage levels in native Spark. And CSAS can reduce 66.7% time compared to M_M which are the default scheme in Spark.

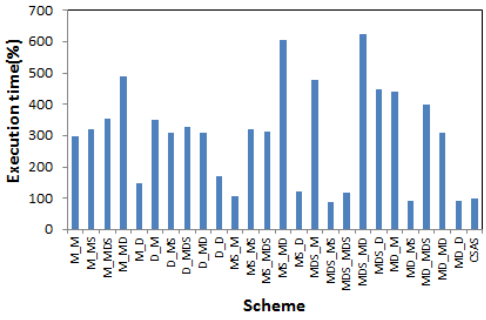

Fig. 3. Overall performance.

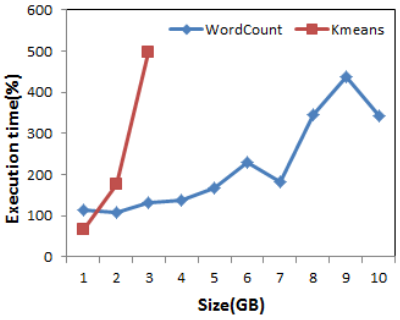

Fig. 4. Performance in different data sizes.

Fig. 4 shows the compare of performance between CSAS and Spark native system which under different input sizes. In the experiment, we set all cache RDDs' storage level to MEMORY_ONLY. For WordCount, CSAS can reduce 8.1-77.2% time compared to M_M in different input sizes. And for Kmeans, Spark fails when input size bigger than 4GB during using the default storage level, but CSAS can work well.

Acknowledgments. Jie Tang is the corresponding author of this paper. This work is supported by South China University of Technology Start-up Grant No. D61600470, Guangzhou Technology Grant No. 201707010148 and National Science Foundation of China under grant No. 61370062.

References

1. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., and Stoica, I.: Resilient Distributed Datasets: a Fault-tolerant Abstraction for In-Memory Cluster Computing'. In: Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, San Jose, CA, pp. 2-2(2012)
2. Choi, I.S., W. Yang, and Y.S. Kee.: Early Experience with Optimizing I/O Performance using High-performance SSDs for In-Memory Cluster Computing. In: Proceedings of IEEE International Conference on Big Data (Big Data 2015), (2015)
3. Geng, Y., et al.,: LCS: An Efficient Data Eviction Strategy for Spark. International Journal of Parallel Programming, pp. 1-13(2016)
4. BigDataBench, http://prof.ict.ac.cn/BigDataBench/