Locality-based Partitioning for Spark

Xia Yuchong[1] , Yang Fangfang[1]

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

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

Keywords: Spark, shuffle, locality, data skew.

Abstract: Spark is a memory-based distributed data processing framework. Lots of data is transmitted through the network in the shuffle process, which is the main bottleneck of the Spark. Because the partitions are unbalanced in different nodes , the Reduce task input are unbalanced. In order to solve this problem, a partition policy based on task local level is designed to balance the task input. Finally, the optimization mechanism is verified by experiments, which can alleviate the data-skew and improve the efficiency of the job process.

1. Introduction

Apache Spark[1] is one of the popular parallel computing frameworks, which has the characteristics of fast, general, simple and so on. Due to the use of memory based computing, Spark computing speed is 100 times faster than Hadoop[2] MapReduce in memory, or 10x faster on disk. Aiming at the poor efficiency of the applications of iterative machine learning algorithms and interactive data mining in the MapReduce [3,4] computational frameworks, spark proposes a distributed memory abstraction, which is called resilient distributed datasets(RDD). It not only retains the scalability, fault tolerance and compatibility of MapReduce, but also makes up for the deficiency of MapReduce. Compared with other parallel computing frameworks, Spark is more efficient for large scale data processing.

With more and more business area used Spark, the framework inherent shortcomings has gradually came out. Spark framework produce a large amount of temporary files, unbalanced dataset and the pulled too much data during the shuffle process, data-skew, which greatly affects the efficiency of Spark performance. Data-skew are common in Spark, which is the main reason to restrict the execution of the job. Therefore, the optimization of shuffle mechanism becomes an urgent problem in Spark.

2. Related work

There has been a lot of research on the shuffle mechanism of the Spark framework. As many MapReduce implementations have been used to process massive data, shuffle phase is proved to be one of the most threats to the Spark. As there are too many small files in the shuffle process, which seriously affect the disk I / O efficiency, Shuffle File Consolidation[5] has been proposed to reduce the amount of temporary files generated during shuffle by combining the temporary files; In addition, in order to solve the problem partition non-uniformity in partitioning strategy, Benjamin Gufler[6] proposed Fine Partitioning and Dynamic Fragmentation algorithm to re-partition the output of Map task so that each Reduce task partitions are roughly the same and alleviate the data skew problem; In [7], Chen Yingzhi Proposed an adaptive memory scheduling algorithm based on the overflow history to solve the memory overflow error in the shuffle phase caused by the imbalance of task memory requirement; In [8] [9], the adaptive strategy of SCID (Splitting and Combination algorithm) and SASM (Spark Adaptive Skew Mitigation) are proposed respectively, and the data-skew problem is mitigated by data migration; Mosharaf Chowdhury [10] proposed Orchestra scheduling framework, the overall network scheduling strategy to improve the network utilization during shuffle phase.

Most of the above researches focus on scheduling and memory optimization, while ignoring the impact of network on shuffle. In the presence of partitioning skew, the existing shuffle strategy encounters the problems of long intermediate data shuffle time and noticeable network overhead. with the presence of partitioning skew, the blindly hash-partitioning is inadequate and can lead to (1) network congestion caused by the huge amount of shuffled data, (2) unfairness of Reduce tasks inputs, To overcome these challenge ,we propose a partitions algorithm based on the location of the intermediate data. What's more we have developed an novel approach to balance the Reduce task inputs.

3. RDD and partitions

RDD (Resilient Distributed Datasets) is a highly constrained shared memory model. And RDDs are motivated by two types of applications that current computing frameworks handle inefficiently: iterative algorithms and interactive data mining tools. RDD is a read-only, partitioned collection of records. RDD can only be created based on the execution of deterministic operations on the dataset in the stable physical store and other RDDs. Transformations and actions are the main RDD operations in Spark. In addition to these operators, users can ask for an RDD to persist. Each RDD implements 5 internal interfaces: *Operation(),Partitions(),preferredLocations(p),Dependencies(),Iterator(p, parentIters), Partitioner().*

There are two dependencies between RDD: narrow dependency and wide dependency. Narrow dependency, each partition of parent RDD is used by at most one partition of child RDD. And for the wide dependency, multiple child partitions may depend on the same parent RDD. First, narrow dependency allows all parent partitions to be calculated on a cluster node in a pipelined way(such as map filter). And wide dependency will need to first calculate all the parent partition and be shuffled across the node, which is similar to the MapReduce. The process of RDD is shown in Fig 1.

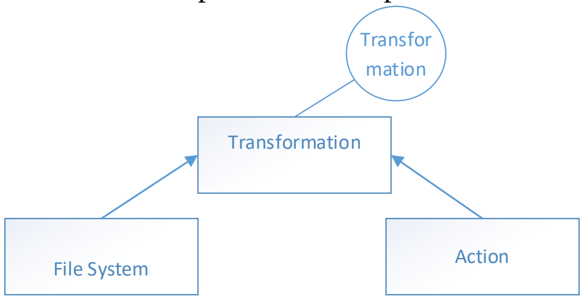

Fig. 1 RDDs process

In MapReduce framework, the shuffle phase is a bridge between Map and Reduce. The output of Map task is transmitted to Reduce task through the shuffle. The default partitioning algorithm in Spark is Hash(*hash (Hash code (Intermediate-key) Modulo Partitions)*),and different key represents different partitions.

4. Local-based partitioning

4.1 Problem analysis

(1) Data-skew

After MapReduce became a popular parallel framework, data skew has become one of the key factors affecting the efficiency of job. As described in section 3, during the shuffle phase, each Reduce task must fetch some elements from every Map task. As the size of partitions depends on the number of relevant key/value tuples, data skew will give rise to the imbalance among different Reduce task because of the keys dispatching based on hashing algorithm.

Since the concept of Stage is used in Spark, the child Stage will not be committed until the parent Stage is completed. And we know that Spark assigns one task every partition and each worker can process one task at a time. So, the imbalanced data of Reduce task input among different node, not only delayed task execution time, but also less resource utilization. This is shown in Fig 2.

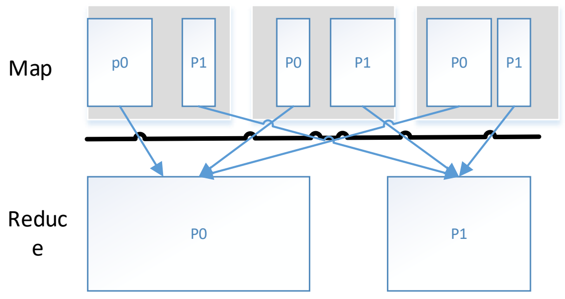

Fig. 2 Data skew

(2) Data Locality

Locality [11] is to schedule the task on a machine that stores its inputs. With the delay scheduling, Spark can automatically adjust its own local match policy based on the submitted time of previous task. As it goes, "transfer data is not as good as transmission calculation", task is started preferentially at the location of the data store, which improves the local preferment level of task during the scheduling process. This paper[10] shows that the network I/O takes up more than 33% of the whole shuffle process. Task locality is one of the main performance indicators of Spark.

4.2 Our approach

In this paper, we use $I \subseteq k \times v$ to represent the intermediate data from map tasks, where k and v are respectively the sets of keys and values. We define that a cluster is formalized as a subset containing all key/value tuples with a specific key: $C_k = \{(k,v) \in I\}, k \in K, v \in V$. The partition for intermediate data is determined by the partitioning algorithm: $F : K \to \{1,\ldots,p\}$.The intermediate results are split into p partitions according to the keys of the tuples. We define the partition as: $P(j) = \bigcup_{k \in K : F(k) = j} = C(k)$.

With the delay scheduling [12], we can get the global distribution information of the data before the shuffle and then we partition the data. The architecture is shown in Fig3.

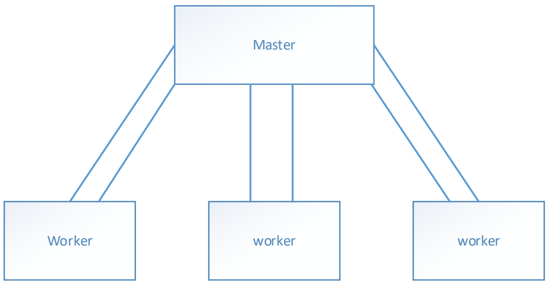

Fig. 3 Architecture of Spark

As depends on multiple child partitions, shuffledRDD doesn’t has the preferredLocation. In order to improve the local level of the task, we schedule the task to the node which achieves the largest partition. For a given K, we calculate the variance of the amount of data distributed on each node. Where n is the number of node, and $Mean_i$ represents the mean of the i-th datasets. For a specific key to achieve the best locality, we select the node with maximum frequency.

During the partition process, we use the data cluster as the unit. Because this can not only guarantee a task to deal with a partition, but also avoid the large partitions. For a specific key when the variance of the partition is the smallest, it is accepted. After selecting the node, it moves on to the next key.

Since the amount of data in the Spark is very large, the number of datasets (the number of keys) will be large. All using the above partition strategy, will take up too much time. For the large K, there is no need to partition, we labeled as a separate partition. And we partition the small k according the default partition. The partition strategy is as shown in algorithm 1.

```
Input: K(the set of keys), N(the number of nodes)
Output: $p(k_i, n_j)$//partitions result
(1) mean = Calculate_Mean(K)
(2) FOR $k_i$ in K:
(3)     IF $k_i$>=2*mean
(4)         j = FindNode($k_i$)
(5)         $p(k_i,n_j)$
(6)     ELIF $k_i$<=min:
(7)         j=Hash($k_i$)
(8)         $p(k_i,n_j)$
(9)     ELSE:
(10)        temp=max ,n=0
(11)        FOR j in range(0,N):
(12)            $va_j$=Calculate_Variance($k_i$,p)
(13)            IF $va_j$<temp
(14)                temp = $va_j$, n=j
(15)        END FOR
(16)        $p(k_i,n_j)$
(17) ENF FOR
```

Fig. 4 Local-based partitioning

5. Experimental evaluation

Our experiments are performed on a cluster consisting of six nodes running Spark 1.5.0 with a separate master node. All these nodes have 4-core 2.49GHZ CPU, 24GB of RAM, and one 1.2T disk driver. And all of nodes are connected by 1Gbps Ethernet network. The operating system on these nodes is Centos, with kernel version 2.6.32-642, and the JDK version is jdk-8u91-linux-i586.Storage nodes use HDFS(Hadoop 2.6) as DFS and YARN as the resource negotiator. And the initial distribution of data is shown in the following Table 1

	K0	K1	K2	K3	K4	K5	K6	K7	K8
Node1	15	15	10	6	19	4	1	5	0
Node2	25	15	7	7	5	2	3	10	1
Node3	13	24	15	10	9	1	1	1	1

In the experiment, we compare Locality-based partitioning to Hash partitioning .The results shows in table 2 and table 3.

Table 2 Hash partitioning

	K0	K1	K2	K3	K4	K5	K6	K7	K8
Node1	53	0	0	23	0	0	5	0	0
Node2	0	54	0	0	33	0	0	16	0
Node3	0	0	32	0	0	7	0	0	2

Table 3 Localit-based partitioning

	K0	K1	K2	K3	K4	K5	K6	K7	K8
Node1	53	0	0	23	0	0	5	0	0
Node2	0	54	0	0	33	0	0	16	0
Node3	0	0	32	0	0	7	0	0	2

As can be seen in the tables, each cluster can only be assigned to one partition.Using Locality-based partitioning, the keys which will be locally partitioned on each of the three nodes are 44%, 50%, and 46% respectively, with an average of 47%, and 50% improvement of the data locality in Hash partitioning as shown in table1.Subsequently, Locality-based partitioning reduces the amount of data transfer by 24%, 120 keys were shuffled. More importantly, Locality-based partitioning achieved very close to optimal data distribution of reduces' inputs, 74, 74, and 77 respectively.

Experiment verifies that the balanced-schedule strategy can improve shuffle efficiency and reduce the job completion time.

6. Conclusions

By identifying the shuffle phase bottlenecks specific to spark, this paper propose the Locality-based partitioning. Experimental results show that the Locality-based partitioning can deliver a better improvement over Spark in different job. Stragglers are a common occurrence in clusters with many clusters reporting significantly slow tasks despite many prevention and speculation solutions. A Facebook trace from 2010[15] shows that less than 60% of tasks achieve locality even with three replicas. In the future work, we will focus on the stragglers and improve the task locality to minimizes the time it takes to transfer all remote inputs.

Acknowledgments

This work was financially supported by Chongqing University of Posts and Telecommunications Doctoral Foundation (NO.A2015-17).

References

[1] Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing[C]//Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation. USENIX Association, 2012: 2-2.
[2] Apache Spark[EB/OL]. https://spark.apache.org/
[3] Chen Q, Liu C, Xiao Z. Improving mapreduce performance using smart speculative execution strategy[J]. IEEE Transactions on Computers, 2014, 63(4): 954-967.
[4] Gunarathne T, Zhang B, Wu T L, et al. Scalable parallel computing on clouds using Twister4Azure iterative MapReduce[J]. Future Generation Computer Systems, 2013, 29(4): 1035-1048.
[5] Davidson A, Or A. Optimizing shuffle performance in spark[J]. University of California, Berkeley-Department of Electrical Engineering and Computer Sciences, Tech. Rep, 2013.
[6] Benjamin G, Nikolaus A, Angelika R, et al. Handling data skew in mapreduce[C]//Proc. Int. Conf. Cloud Comput. Serv. Sci. 2011: 574-583.
[7] Chen Yingzhi. Analysis and optimization of memory scheduling algorithm in Spark shuffle [D]. Zhejiang University, 2016 .
[8] Tang Z, Zhang X, Li K, et al. An intermediate data placement algorithm for load balancing in Spark computing environment[J]. Future Generation Computer Systems, 2016.
[9] Yu J, Chen H, Hu F. SASM: Improving spark performance with Adaptive Skew Mitigation[C]//Progress in Informatics and Computing (PIC), 2015 IEEE International Conference on. IEEE, 2015: 102-107.
[10] Chowdhury M, Zaharia M, Ma J, et al. Managing data transfers in computer clusters with orchestra[C]//ACM SIGCOMM Computer Communication Review. ACM, 2011, 41(4): 98-109.
[11] Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters[J]. Communications of the ACM, 2008, 51(1): 107-113.
[12] Zaharia M, Borthakur D, Sen Sarma J, et al. Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling[C]//Proceedings of the 5th European conference on Computer systems. ACM, 2010: 265-278.