International Journal of Parallel Programming (2020) 48:941–956
https://doi.org/10.1007/s10766-020-00657-z

Handling Data Skew for Aggregation in Spark SQL Using Task Stealing

Zeyu He[1] · Qiuli Huang[1] · Zhifang Li[1] · Chuliang Weng[1]

Received: 6 August 2019 / Accepted: 11 November 2019 / Published online: 25 March 2020
© Springer Science+Business Media, LLC, part of Springer Nature 2020

Abstract
In distributed in-memory computing systems, data distribution has a large impact on performance. Designing a good partition algorithm is difficult and requires users to have adequate prior knowledge of data, which makes data skew common in reality. Traditional approaches to handling data skew by sampling and repartitioning often incur additional overhead. In this paper, we proposed a dynamic execution optimization for the aggregation operator, which is one of the most general and expensive operators in Spark SQL. Our optimization aims to avoid the additional overhead and improve the performance when data skew occurs. The core idea is *task stealing*. Based on the relative size of data partitions, we add two types of tasks, namely *segment tasks* for larger partitions and *stealing tasks* for smaller partitions. In a stage, stealing tasks could actively steal and process data from segment tasks after processing their own. The optimization achieves significant performance improvements from 16% up to 67% on different sizes and distributions of data. Experiments show that involved overhead is minimal and could be negligible.

Keywords In-memory computing · Spark SQL · Aggregation · Data skew

✉ Zeyu He
zyhe@stu.ecnu.edu.cn

Qiuli Huang
qiulihuang@stu.ecnu.edu.cn

Zhifang Li
zhifangli@stu.ecnu.edu.cn

Chuliang Weng
clweng@dase.ecnu.edu.cn

[1] School of Data Science and Engineering, East China Normal University, Shanghai, China

1 Introduction

In recent years, as memory capacity continues to increase and DRAM price continues to decrease, the distributed in-memory computing systems represented by Apache Spark [22] have appeared. Spark provides an analytical library named Spark SQL [2] to handle structured and semi-structured data. This paper focuses on the aggregation operator in Spark SQL. It is one of the most common and expensive operators in data processing systems [4,18]. In Spark SQL, the aggregation operator adopts the classic two-phase implementation [14], which is expressed as two *stages*. In the first stage, *tasks* read data from partitions and locally execute the partial aggregation to reduce the input size transferred to the next stage. After the first stage is completed, tasks in the second stage will fetch data from the previous stage and perform the final aggregation. The partial aggregation and the final aggregation can be treated as two reduce operations.

Caching data into memory in advance can significantly speed up the processing of aggregation. However, in the course of our practice, we found that the performance of aggregation largely depends on data distribution even after caching. *Resilient Distributed Dataset* (RDD) [28] is Spark's core data abstraction. In an RDD, each partition is the basic unit of parallelism. Spark assigns one task for each partition. When data skew occurs, most data is distributed in specific partitions. In the first stage of aggregation, tasks for these larger partitions will be finished later than tasks for other partitions. They will become the performance bottleneck of aggregation because most data is aggregated in this stage.

To prevent the data skew problem, users often need to have sufficient prior knowledge of data to distribute it uniformly, which is unpractical in reality. In Spark SQL, the classic method to handle data skew is using the repartition operator to redistribute data in a round-robin fashion. Although the repartition operator could increase the performance of aggregation, it is an extra operation that causes much unnecessary overhead since it needs to access and shuffle all the data. This operation moves large amounts of data across the cluster, which leads to serious performance degradation. Therefore, we should avoid the operation.

In this paper, we propose a dynamic execution optimization to handle data skew for aggregation operators. It is transparent to users and improves the performance with little additional overhead. First, we design an algorithm to detect data skew based on the information about partitions provided by Spark. According to the relative size of partitions, we label tasks for the larger ones as *segment tasks* and label tasks for the smaller ones as *stealing tasks*. If data is skewed, stealing tasks will steal data from segment tasks to process after they have processed their own data. To allow multiple tasks run concurrently on partitions, segment tasks process data in units of fine-grained *segments* by logically splitting their partitions. Our optimization works in the first stage of aggregation. Traditional approaches to handling data skew by sampling [3,9,10] and repartitioning [11,12] often incur additional overhead. The optimization we proposed in this paper not only avoids the overhead, but also utilizes idle resources of stealing tasks which are finished earlier than segment tasks. In this way, it balances the computational burden between segment tasks and stealing tasks and eventually improves the performance when data skew occurs.

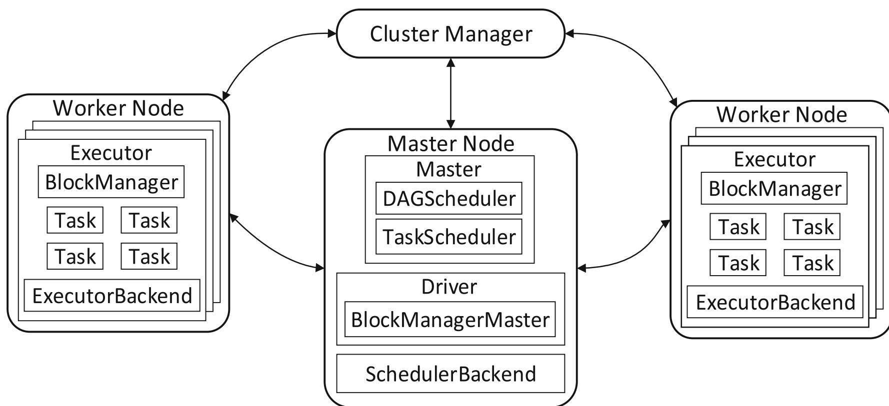

Fig. 1 Spark architecture

Overall, we make the following contributions in this paper:

- We find that aggregation operators subject to different performance degradation on varying degrees of data skew. According to the partition information provided by Spark, we design an algorithm to detect data skew and classify partitions into the larger ones and the smaller ones. Compared to traditional approaches, it avoids the overhead of sampling.
- Segment tasks for larger partitions process data in units of fine-grained segments by logically splitting partitions. Stealing tasks for smaller partitions actively steal and process data from segment tasks after processing their own data. They are implemented in a stage to prevent repartitioning data.
- Our proposal improves the performance with little additional overhead in the presence of data skew. From the evaluation results, it increases the performance for aggregation ranged from 16% to 67%.

The rest of the paper is organized as follows. Section 2 provides background knowledge and discusses the problem. To solve the problem, we propose dynamic execution optimization in Sect. 3. Section 4 expands the optimization in detail. Section 5 evaluates the effectiveness of our optimization by experiments. Related works are revisited in Sect. 6, and conclusions are given in Sect. 7.

2 Background and Motivation

In this section, we provide the necessary background regarding Spark and discuss the problem we need to solve.

2.1 Spark Overview

A Spark cluster consists of one master node and multiple worker nodes. Each node has its components that play different roles in the cluster (see Fig. 1).

In native Spark, it uses the *Directed Acyclic Graph* (DAG) to model the dependency of RDDs. *DAGScheduler* creates stages based on DAG, generates tasks for each stage and submits them to *TaskScheduler* for processing. Each task retains information about the partition it needs to process. TaskScheduler distributes the received tasks to *executors* located in worker nodes through different kinds of *SchedulerBackend*. A worker node has one or more executors. Each executor has its own *BlockManager*, which stores data and provides a data access interface for tasks running on it. Resource allocation is performed by the master node's SchedulerBackend, which allocates the computing resources of executors for every task before execution. Tasks will take up these resources until the end of their stage.

2.2 Problem Statement

To improve the performance, we tend to cache data in memory as partitions in advance. However, the performance gains of caching depend on data distribution. When data is not uniformly distributed in partitions, it will have a significant impact on performance, as shown in Fig. 2. The workload is TPC-H [24]. The size of the dataset is 7.8 GB and detailed configurations can refer to Sect. 5.

Spark generates two stages for aggregation. When data skew occurs, we use the repartition operator to handle data skew and redistribute data uniformly. It requires Spark to create a new stage to handle it. The stage is executed before the aggregation. In Fig. 2, if data distribution is uniform, the aggregation operator spends 1.3 s. However, the cost is up to 5.2 s when data skew occurs. Repartitioning takes 6 s to increase the performance of the aggregation operator to 1.3 s. Overall, the total execution time is 7.3 s. Therefore, repartitioning is an expensive operation we should avoid.

The main reason for the performance degradation of the aggregation operator is presented in Fig. 3. Suppose that there are two worker nodes, and each node has one executor. Every executor in worker nodes caches two partitions. The degree of filling in partitions indicates the size of data. DAGScheduler generates tasks for every stage and simply assigns one task for each partition. When data skew occurs, the workload of tasks will be different. In Stage1, tasks read data from partitions (in RDD1) and locally execute the partial aggregation (in RDD2) to reduce the input size transferred to Stage2. After Stage1 is completed, tasks in Stage2 will fetch data from RDD2 (in RDD3) and perform the final aggregation (in RDD4). Since most data is aggregated in RDD2, Stage1 has the largest computational overhead in the aggregation operator. In Stage1, Task2 and Task3 have more data to process. They are finished later than Task1 and Task4 because computing resources allocated for each task are fixed. Overloaded tasks like Task2 and Task3 will become the performance bottleneck.

3 The Design of Optimization

In this section, we propose a dynamic execution optimization for aggregation, which aims to improve the performance and avoid the additional overhead when data skew occurs. It works in the first stage of aggregation. The core idea of our optimization

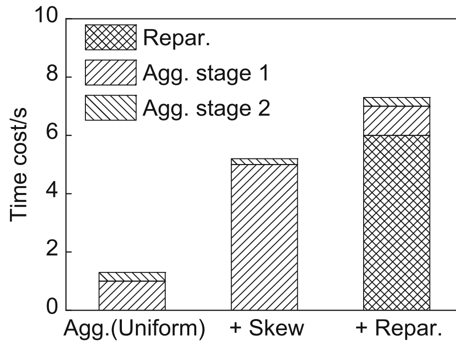

Fig. 2 Performance profiling of aggregation in native Spark

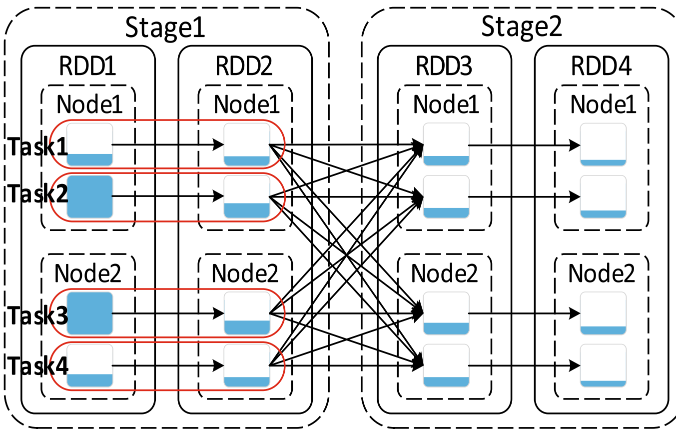

Fig. 3 Execution of aggregation in native Spark

is task stealing. It consists of three steps: (1) add two types of tasks based on the relative size of partitions, namely *segment tasks* for larger partitions and *stealing tasks* for smaller partitions; (2) make segment tasks process data in units of segments by logically splitting their partitions; and (3) let stealing tasks steal and process data from segment tasks after stealing tasks process their partitions. Compared to processing data in units of partitions, Step (2) allows multiple tasks to run on partitions concurrently.

Based on the architecture of native Spark, we implement our optimization as *Spark-M* and modify the components in color, as shown in Fig. 4.

DAGScheduler-M detects data skew in the first stage of aggregation and assigns one *Task-M* to each cached partition. *Task-M* has three categories namely ordinary task, segment task and stealing task. They have their own execution mechanism and access their partition via *BlockManager-M*.

The aggregation operator in Spark-M is executed as shown in Fig. 5. In RDD2, once Task1 and Task4 finish processing their partitions, they will steal data from Task2 and Task3 to aggregate, respectively. It is represented by dashed lines. In our optimization, all stealing tasks will actively steal and process data from partitions that are processed

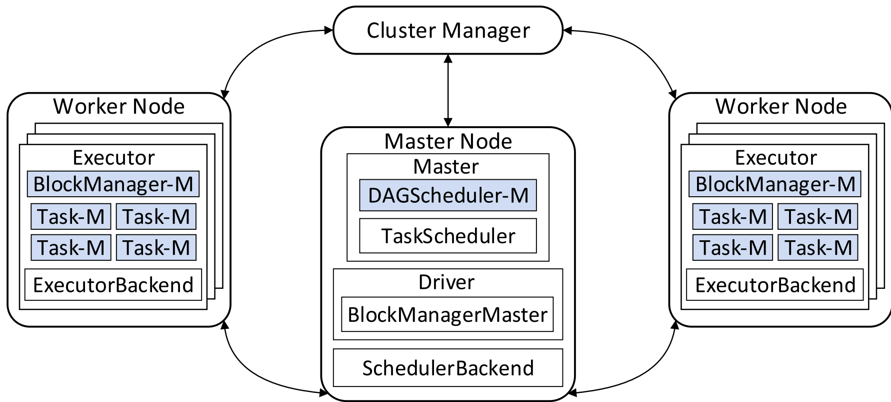

Fig. 4 Spark-M architecture

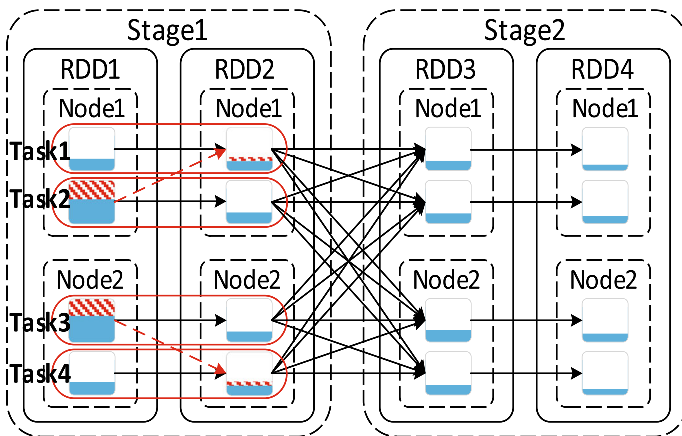

Fig. 5 Execution of aggregation in Spark-M

by segment tasks. It harnesses idle resources of stealing tasks and reduces the workload of segment tasks. In this way, the optimization avoids the overhead of repartitioning. Besides, it is transparent to users so that they do not need to know the characteristics of the distribution.

4 Implementation

This section details the implementation of our dynamic execution optimization from three aspects.

4.1 Partition Classification Algorithm

To detect data skew in every executor and classify partitions, we design Algorithm 1 in DAGScheduler-M based on partition information. The information is fetched from

a Key-Value (KV) store in Spark, which collects information about task execution and partition storage.

```
Algorithm 1: Partition classification algorithm
Input: Information from the KV store about partitions in every executor, hostToCachedParts; A
       factor to detect data skew, φ;
Output: Larger partitions that need to be segmented, splittingParts; Dependencies between
        smaller partitions and larger partitions, smallPartsToLargeParts;
1  foreach (host, parts) in hostToCachedParts do
2      sortedParts = sortByPartSize(parts);
3      rearPoint = sortedParts.size - 1;
4      if sortedParts(rearPoint).size/sortedParts(0).size ≥ φ then
5          for i=0:sortedParts.size do
6              if sortedParts(rearPoint).size/sortedParts(i).size ≥ φ then
7                  smallPartsToLargeParts.insert(sortedParts(i), sortedParts(rearPoint));
8                  splittingParts.insert(sortedParts(rearPoint));
9                  rearPoint -= 1;
10             else
11                 for j=sortedParts.size–1:rearPoint do
12                     if sortedParts(j).size/sortedParts(i).size > φ then
13                         smallPartsToLargeParts.insert(sortedParts(i),  sortedParts(j)) ;
14                         rearPoint = j - 1 ;
15                         break ;
```

According to the partition size, all partitions on every executor are sorted in an array named *sortedParts* in ascending order (line 2). We detect data skew by the size of the smallest partition and the largest partition (lines 3–4). ϕ is a factor to determine whether the data is skewed, with a default value of 1024. A larger ϕ generates less stealing tasks and wastes resources. A smaller ϕ makes stealing tasks fail to steal data, because most segment tasks have finished processing their data when stealing tasks finish their works and try to steal data from segment tasks. If data skew is detected, we establish dependencies named *smallPartsToLargeParts* from smaller partitions to larger partitions according to the size of partitions (lines 5–15). The *splittingParts* stores all the larger partitions that need to be split.

When DAGScheduler-M creates tasks for each stage, it will execute Algorithm 1 to check the degree of data skew. If a task's partition is a larger partition in *splittingParts*, the task is labeled as a segment task. If a task's partition is a smaller partition in *smallPartsToLargeParts*, the task is labeled as a stealing task. Tasks for partitions that are not in *smallPartsToLargeParts* keep the execution mechanism of native Spark, namely ordinary tasks. Besides, DAGScheduler-M will find and assign one larger partition to every stealing task based on *smallPartsToLargeParts*. It means that each stealing task keeps information about two partitions. One is the smaller partition it needs to process, and the other is the larger partition it needs to steal. In this way, a stealing task can steal and process data from the larger partition after processing its

smaller partition. It is based on a basic observation that tasks for larger partitions are often finished later than tasks for the smaller ones.

4.2 Segment Task

In Spark, the executing units of tasks are partitions. It is not suitable for multiple tasks running concurrently on the same partition. Therefore, for segment tasks, we logically split their larger partitions into multiple fine-grained segments and make them process data in units of segments.

Every cached partition in BlockManager is an array including multiple *CachedBatches*. The default size of a CachedBatch is 10000 rows and can be specified by the Spark configuration. In our BlockManager-M, we make a new class named $cachedPartition$ for every larger partition. The class is unique in every stage and is created when the first task in a stage accesses the partition. It guarantees the correctness of applications that concurrently access the partition.

```
private case class cachedPartition() {
    values = array[CachedBatch]
    frontIndex = 0
    rearIndex = values.size
    jumpNum = if (values.size % k != 0) {
        size / k + 1
    } else {
        size / k
    }
    done = false
}
```

k is the number of segments, with a default value of 10. A larger k increases the overhead of function calling, while a smaller k cannot decrease the workload of segment tasks. The attribute $done$ is a flag that tells all tasks running on the partition that data has been processed. The attribute $values$ stores data from a larger partition. The $frontIndex$, the $rearIndex$ and the $jumpNum$ are integers which are used to read CachedBatches from $values$.

Based on the class $cachedPartition$, we design a split processing algorithm for segment tasks. First, we check if the segment task is the first task in a stage to access its partition (lines 1–2). Next, the task will read and process data in units of segments until all data is processed (lines 3–17).

The $cachedParts$ stores all $cachedPartitions$ of a stage. If a segment task does not find the $cachedPartition$ in $cachedParts$, it is the first task to access the $lPart$ and needs to construct a new $cachedPartition$ in $cachedParts$ (lines 1–2). The lock is used to make sure tasks running on the $cachedPartition$ are synchronized (line 5). With the help of $frontIndex$, $jumpIndex$ and $rearIndex$, the segment task directly reads and processes data from the $cachedPartition$ in units of segments rather than splitting them first (lines 7–11). Data is processed from $frontIndex$ to $rearIndex$. When the remaining data is not enough to form a segment, it means that this is the last segment the task needs to process. In this case, the segment task will read the segment

and tell other stealing tasks running on the $cachedPartition$ that the partition has been processed (lines 13–14).

4.3 Stealing Task

Every stealing task holds the information about a smaller partition and a larger partition. After stealing tasks process their smaller partitions, they could directly find the larger partitions they need to steal without additional overhead. We design a data stealing algorithm for stealing tasks. Every stealing task processes data from its smaller partition first (line 1). Then the task will steal and process data from the larger partition (lines 2–14).

Similar to the segment task, the stealing task constructs a $cachedPartition$ in $cachedParts$ if it is the first task in a stage to access the $lPart$ (lines 2–3). When the segment task has not finished processing the $cachedPartition$, the stealing task will steal data from it (line 5). As the amount of processed data increases, the $rearIndex$ is gradually decreased and the $frontIndex$ is gradually increased. Therefore, the $stealSegment$ is also decreased (line 8). δ determines how much data would be stolen based on the remaining data. A too small δ cannot balance the workload between stealing tasks and segment tasks, while a too large δ makes stealing tasks become the new bottleneck. As far as our practice, 3 is the most efficient value. To prevent stealing tasks from becoming the new bottleneck, we also make them only steal data once on the same partition. Unlike the processing order of segment tasks, stealing tasks process data in the $cachedPartition$ from $rearIndex$ to $frontIndex$. It minimizes the conflicts between stealing tasks and segment tasks. When the $cachedPartition$ has been completely processed by the segment task, the stealing task only processes its $sPart$ and saves the processing result. Otherwise, the stealing task will save the processing results from the $sPart$ and part of the $lPart$ (line 14).

To ensure the concurrent updates of *cachedPartition*, Algorithm 2 and Algorithm 3 both lock it when reading data and unlock it before processing. Because structured data cached by Spark SQL is deserialized, the time of reading data almost could be negligible. In this way, the overhead of synchronization is minimal.

5 Experiments

In this section, we evaluate the performance of the aggregation operator with our dynamic execution optimization on different datasets and degrees of data skew. We implemented our optimization in Spark-2.3.2. It is deployed on a cluster connected by a 10 GigE switch, of which 3 nodes are used for worker nodes, and 1 node is used for the master node. Each node has two 2.1 GHz Intel Xeon Silver 4110 8-core processors, 156 GB memory, and runs 64-bits CentOS 7.5.1804. We use two subqueries from Q18 and Q15 in TPC-H workload:

(1) *SELECT orderkey, SUM(quantity) FROM lineitem GROUP BY orderkey;*
(2) *SELECT suppkey, SUM(extendedprice*(1-discount)) FROM lineitem WHERE shipdate ≥ '1994-01-01' GROUP BY suppkey;*

Algorithm 2: Split processing algorithm

```
Input: The larger partition that the task needs to process, lPart; A map from partition Id to
       cachedPartition in a BlockManager-M, cachedParts;
1  if cachedParts.contains(lPart.Id) == false then
2      constructing a new cachedPartition in cachedParts based on lPart;
3  part = cachedParts.find(lPart.Id);
4  while part.done == false do
5      part.lock();
6      cachedBatches = part.values;
7      if part.frontIndex + part.jumpNum < part.rearIndex then
8          segment = cachedBatches.slice(part.frontIndex, part.frontIndex + part.jumpNum);
9          part.frontIndex = part.frontIndex + part.jumpNum;
10         part.unlock();
11         processing data from segment;
12     else
13         segment = cachedBatches.slice(part.frontIndex, part.rearIndex);
14         part.done = true;
15         part.unlock();
16         processing data from segment;
17 saving the processing result and finishing the task;
```

Algorithm 3: Data stealing algorithm

```
Input: The smaller partition that the task needs to process, sPart;
       The larger partition that the task needs to steal, lPart;
       A map from partition Id to cachedPartition in a BlockManager-M
       cachedParts;
       A factor to control the size of data stealing, δ;
1  finding the sPart from BlockManager-M and processing data from it;
2  if cachedParts.contains(lPart.Id) == false then
3      constructing a new cachedPartition in cachedParts based on lPart;
4  part = cachedParts.find(lPart.Id);
5  if part.done == false then
6      part.lock() ;
7      cachedBatches = part.values;
8      stealJumpNum = (part.rearIndex - part.frontIndex) / δ ;
9      if stealJumpNum > 0 then
10         stealSegment = cachedBatches.slice(part.rearIndex - stealJumpNum, part.rearIndex);
11         part.rearIndex = part.rearIndex - stealJumpNum ;
12         part.unlock();
13         processing data from stealSegment;
14     else
15         part.unlock();
16 saving the processing result and finishing the task;
```

Query (1) includes an aggregation operator with the high-cardinality attribute *orderkey*. The *orderkey* has a large number of different data values. In the first stage, data is grouped by the values. It means that data transferred to the second stage is large. Query (2) consists of a filter operator and an aggregation operator with the low-cardinality attribute *suppkey*. Almost all computations happen in the first aggregation

Table 1 Data distribution

	Node 1			Node 2			Node 3		
	Small	Medium	Large	Small	Medium	Large	Small	Medium	Large
Distr. 1	30	0	2	29	0	3	30	0	2
Distr. 2	18	14	0	18	14	0	19	13	0
Distr. 3	32	0	0	32	0	0	32	0	0

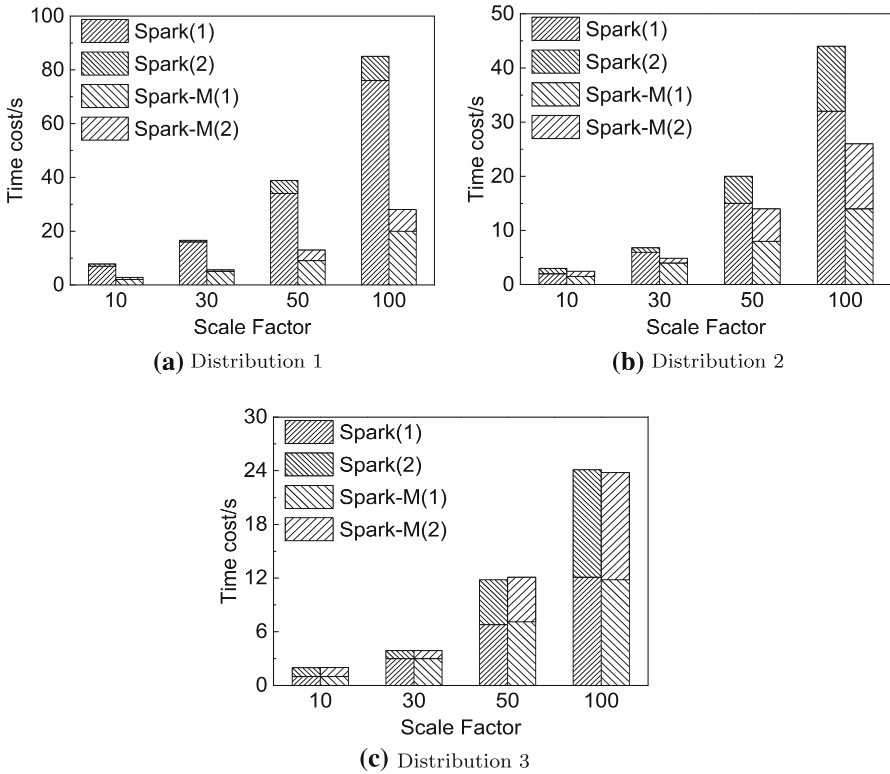

Fig. 6 Performance evaluation in different distributions for Query (1)

stage. It is a compute-intensive query. Considering the size of datasets, we configure 80 GB of memory for each Spark executor. Stealing tasks may open too many files during the shuffle phase, which could exceed the maximum of file handles set by the Linux kernel. According to the number of CPUs in our cluster, we set the Spark configuration *spark.sql.shuffle.partitions* from 200 to 96.

Three sets of experiments with different data distributions and datasets for each query are designed. Data distribution is shown in Table 1. There are 32 partitions cached on each node. They are classified into three categories: small, medium and large partitions. The small partition indicates that the size of it accounts for 0% to 5%

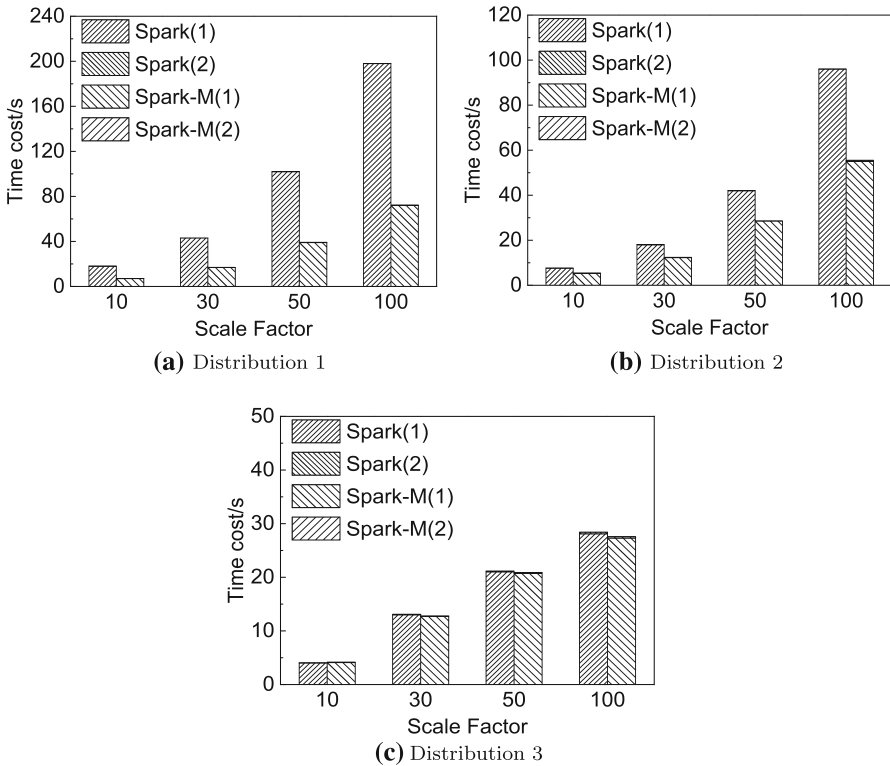

Fig. 7 Performance evaluation in different distributions for Query (2)

of the total data size on the node. The medium partition accounts for 5% to 20% and the large partition accounts for more than 20%.

Distribution 1 simulates the high data skew and most data is distributed in only a few partitions. Distribution 2 has a lower degree of data skew. There are no large partitions in it, which is more common in reality. Distribution 3 is the ideal distribution, where data is uniformly distributed.

Figures 6 and 7 show the performance of the aggregation operator for different datasets under varying data distributions. We use four *lineitem* tables with different sizes, 7.8 GB (scale factor 10–60 M tuples), 23.6 GB (scale factor 30–180 M tuples), 39.5 GB (scale factor 50–300 M tuples) and 79.6 GB (scale factor 100–600 M tuples). Spark in figures is native, which is used for comparison. Spark-M is the modified Spark with our dynamic execution optimization. For each experiment, X(1) and X(2) represent two stages of the aggregation operator, respectively.

As shown in Fig. 6, the performance improvement in Fig. 6a is higher than that in Fig. 6b. It is because a large number of stealing tasks are generated in extreme distribution, and our optimization makes full use of these tasks. In addition to the degree of data skew, the performance improvement that task stealing brings is also related to the size of data partitions. As the total size of data increases, the size of data partitions also increases. It improves the success rate of task stealing. If the partitions assigned

to segment tasks are small, some of segment tasks may have finished processing their partitions when stealing tasks try to steal data from them. Therefore, SF50 and SF100 have a higher speedup than SF10 and SF30 no matter in Distribution 1 or in Distribution 2. Spark-M could achieve the performance improvement of up to 67.06% when the scale factor is equal to 100 in extreme distribution, which is presented in Fig. 6a.

Compared to Query (1), Query (2) has a filter operator and a multiplication computation in SUM(). The filter operator keeps most data, which further increases the computational overhead of each task. For stealing tasks, the success rate of task stealing is also improved because segment tasks need more time to process their data. In Fig. 7, because the low-cardinality attribute *suppkey* aggregates most data in the first stage, almost all the overhead is in this stage. The execution time of the second stage could be negligible. In this case, our optimization has significant performance gains across all distributions and datasets. Figures 6c and 7c illustrate that Spark-M has little impact on performance when data is uniformly distributed. It means that the dynamic execution optimization we proposed in this paper improves the performance with slight overhead.

6 Related Work

In this section, we briefly introduce researches in the following categories: (1) optimizations for aggregation and (2) optimizations for load balancing.

6.1 Aggregation

In recent years, there are a lot of optimizations for aggregation operators. Wang et al. [25] optimized aggregation operators from two aspects in NUMA architectures: the NUMA-aware data partition algorithm and the efficient aggregation algorithm. Although the partition algorithm considers inter-socket and intra-socket load balancing, it still needs to repartition data. Our approach makes idle tasks steal and process part of data from tasks with higher workloads during aggregation. It avoids the overhead of repartitioning. Polychroniou et al. [20] and Jiang et al. [8] utilized SIMD and MIMD instructions to accelerate the processing of aggregation. It optimizes the aggregation operator from the perspective of data parallelism. Müller et al. [18] proposed an aggregation algorithm to dynamically switch hash-based and sort-based aggregation in the process of execution. It increases the performance from the perspective of the algorithm. Culhane et al. [5,6] proposed LOOM, which builds an aggregation tree with some fixed configurations. It requires prior knowledge of data to get the optimal performance. Liu et al. [14] considered similarity between partitions and proposed GRASP. It combines with static network bandwidth information to design a multi-phase execution framework. However, static resource statistics are not accurate and calculating the similarity of partitions needs additional overhead. Therefore, we choose to optimize the performance of aggregation operators from the perspective of data distribution.

6.2 Load Balancing

Load balancing is a classic research topic in parallel computing. Work stealing [1,13, 17,27] is a traditional solution to load imbalance caused by multiple applications. In this way, the whole task in a CPU queue could be stolen to other idle CPUs. Essentially, it is more like task migration. Speculative execution provided by native Spark is based on the idea. Spark periodically checks the running tasks. When the running time of a task is larger than a threshold, Spark will copy and start the task on another idle node. The finish time is based on the first finished one of the two tasks.

However, work stealing could not deal with the load imbalance caused by data skew. It produces a lot of large data partitions and have a large impact on performance [7]. Most prior works which try to avoid and handle data skew are based on data statistics. Okcan et al. [19] and Kwon et al. [9,10] launch procedures to collect statistics before job execution. Chen et al. [3] collect statistics during job execution. Because of expensive sampling and repartitioning, these works often incur additional overhead. Besides, the accuracy of sampling is difficult to guarantee. Liu et al. [16] handle date skew by dynamically adjusting the resource of tasks. Wang et al. [26] introduce a scheduler in every node to dynamically allocate resources for query execution segments in the pipelines, which still incurs additional overhead of scheduler. SkewTune [11,12] detects straggler tasks which process the larger partitions at runtime and repartitions their data. However, SkewTune needs to stop the straggler tasks first when repartitioning data. We detect skewed partitions via the information provided by Spark and propose dynamic execution optimization based on task stealing to handle data skew. Our approach makes tasks with lower workloads actively steal data from larger partitions to achieve load balancing. It avoids the overhead of detecting procedures and repartitioning. In Spark, Bertolucci et al. [21] researched how different data partitioning strategies affect the performance of graph computations. Liu et al. [15] proposed a partition method to handle data skew for Spark Streaming. Zhuo et al. [23] proposed a partitioning strategy to handle data skew for Spark on the shuffle phase. These works rely on sampling. In this paper, we optimize the stage before shuffling, and fetch partition information from a KV store in Spark to avoid sampling data.

7 Conclusions

Data distribution has a crucial impact on the performance of aggregation. When data skew occurs, tasks with higher workloads in the first stage of aggregation become the performance bottleneck. In this paper, we propose a dynamic execution optimization to balance the workload among tasks. It works in the first stage of aggregation and makes tasks for smaller partitions steal and process data from tasks for larger partitions. Our optimization avoids the overhead of sampling and repartitioning. Experiments show that it could achieve significant performance improvements on varying degrees of data skew with little additional overhead. Because we have modified the task scheduling in Spark, the challenge of extending our optimization to general Spark or other Spark libraries is redesigning the splitting scheme based on different data structures of par-

titions. Besides, it can also be extended to other operators and systems to handle data skew, which is one of our future work.

Acknowledgements This research was supported by the National Key Research & Development Program of China (No. 2018YFB1003400).

References

1. Acar, U.A., Chargueraud, A., Rainey, M.: Scheduling parallel programs by work stealing with private deques. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 219–228. PPoPP '13, ACM, New York, NY, USA (2013)
2. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark sql: relational data processing in spark. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1383–1394. SIGMOD '15, ACM, New York, NY, USA (2015)
3. Chen, Q., Yao, J., Xiao, Z.: LIBRA: lightweight data skew mitigation in mapreduce. IEEE Trans. Parallel Distrib. Syst. **26**(9), 2520–2533 (2015)
4. Cieslewicz, J., Ross, K.A.: Adaptive aggregation on chip multiprocessors. In: Proceedings of the 33rd International Conference on Very Large Data Bases, pp. 339–350. VLDB '07, VLDB Endowment (2007)
5. Culhane, W., Kogan, K., Jayalath, C., Eugster, P.: LOOM: optimal aggregation overlays for in-memory big data processing. In: 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14), pp. 13–13. USENIX Association (2014)
6. Culhane, W., Kogan, K., Jayalath, C., Eugster, P.: Optimal communication structures for big data aggregation. In: 2015 IEEE Conference on Computer Communications, pp. 1643–1651. IEEE (2015)
7. Hua, K.A., Lee, C.: Handling data skew in multiprocessor database computers using partition tuning. In: Proceedings of the 17th International Conference on Very Large Data Bases, pp. 525–535. VLDB '91, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1991)
8. Jiang, P., Agrawal, G.: Efficient SIMD and MIMD parallelization of hash-based aggregation by conflict mitigation. In: Proceedings of the International Conference on Supercomputing, pp. 24:1–24:11. ICS '17, ACM, New York, NY, USA (2017)
9. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In: Proceedings of the 1st ACM Symposium on Cloud Computing, pp. 75–86. SoCC '10, ACM, New York, NY, USA (2010)
10. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: A study of skew in mapreduce applications. Open Cirrus Summit **11**, 30 (2011)
11. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skewtune in action: mitigating skew in mapreduce applications. Proc. VLDB Endow. **5**(12), 1934–1937 (2012)
12. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skewtune: mitigating skew in mapreduce applications. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 25–36. SIGMOD '12, ACM, New York, NY, USA (2012)
13. Li, J., Agrawal, K., Elnikety, S., He, Y., Lee, I.T.A., Lu, C., McKinley, K.S.: Work stealing for interactive services to meet target latency. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 14:1–14:13. PPoPP '16, ACM, New York, NY, USA (2016)
14. Liu, F., Salmasi, A., Blanas, S., Sidiropoulos, A.: Chasing similarity: distribution-aware aggregation scheduling. Proc. VLDB Endow. **12**(3), 292–306 (2018)
15. Liu, G., Zhu, X., Wang, J., Guo, D., Bao, W., Guo, H.: SP-partitioner: a novel partition method to handle intermediate data skew in spark streaming. Future Gener. Comput. Syst. **86**, 1054–1063 (2018)
16. Liu, Z., Zhang, Q., Zhani, M.F., Boutaba, R., Liu, Y., Gong, Z.: DREAMS: dynamic resource allocation for mapreduce with data skew. In: 2015 IFIP/IEEE International Symposium on Integrated Network Management, pp. 18–26. IEEE (2015)
17. Merkel, A., Stoess, J., Bellosa, F.: Resource-conscious scheduling for energy efficiency on multicore processors. In: Proceedings of the 5th European Conference on Computer Systems, pp. 153–166. EuroSys '10 (2010)

18. Müller, I., Sanders, P., Lacurie, A., Lehner, W., Färber, F.: Cache-efficient aggregation: hashing is sorting. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1123–1136. SIGMOD '15, ACM, New York, NY, USA (2015)
19. Okcan, A., Riedewald, M.: Processing theta-joins using mapreduce. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data, pp. 949–960. SIGMOD '11, ACM, New York, NY, USA (2011)
20. Polychroniou, O., Raghavan, A., Ross, K.A.: Rethinking SIMD vectorization for in-memory databases. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, pp. 1493–1508. SIGMOD '15, ACM, New York, NY, USA (2015)
21. Ricci, L., Carlini, E., Dazzi, P., Lulli, A.: Static and dynamic big data partitioning on apache spark. In: Conference on Parallel Computing, vol. 27, pp. 489–498. IOS PRESS (2016)
22. Spark homepage. https://spark.apache.org, last accessed 9 May 2019
23. Tang, Z., Zhang, X., Li, K., Li, K.: An intermediate data placement algorithm for load balancing in spark computing environment. Future Gener. Comput. Syst. **78**, 287–301 (2018)
24. The TPC-H benchmark. http://www.tpc.org/tpch, last accessed 10 May 2019
25. Wang, L., Zhou, M., Zhang, Z., Shan, M.C., Zhou, A.: NUMA-aware scalable and efficient in-memory aggregation on large domains. IEEE Trans. Knowl. Data Eng. **27**(4), 1071–1084 (2015)
26. Wang, L., Zhou, M., Zhang, Z., Yang, Y., Zhou, A., Bitton, D.: Elastic pipelining in an in-memory database cluster. In: Proceedings of the 2016 International Conference on Management of Data, pp. 1279–1294. SIGMOD '16, ACM, New York, NY, USA (2016)
27. Wimmer, M., Cederman, D., Träff, J.L., Tsigas, P.: Work-stealing with configurable scheduling strategies. In: Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 315–316. PPoPP '13, ACM, New York, NY, USA (2013)
28. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., 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, pp. 2–2. NSDI'12, USENIX Association, Berkeley, CA, USA (2012)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.