Received: 14 June 2019 | Revised: 15 October 2019 | Accepted: 19 November 2019
DOI: 10.1002/cpe.5637

WILEY

RESEARCH ARTICLE

Handling data skew at reduce stage in Spark by ReducePartition

Wenxia Guo | **Chaojie Huang** | **Wenhong Tian**

School of information and software engineering, University of Electronic Science and Technology of China, Chengdu, China

Correspondence
Wenxia Guo, School of Information and Software Engineering, University of Electronic Science and Technology of China, Chengdu, 610054, China.
Email: 359873769@qq.com

Funding information
National Natural Science Foundation of China, Grant/Award Number: 61672136, 61828202

Summary

As a typical representative of distributed computing framework, Spark has been continuously developed and popularized. It reduces the data transmission time through efficient memory-based operations and solves the shortcomings of the traditional MapReduce computation model in iterative computation. In Spark, data skew is very prominent due to the uneven distribution of input data and the unbalanced allocation of default partitioning algorithm. When data skew occurs, the execution efficiency of the program will be reduced, especially in the reduce stage of Spark. Therefore, this paper proposes ReducePartition to solve data skew problem at reduce stage of Spark platform. First, the compute node samples the local data according to the sampling algorithm to predict the overall characteristics of data distribution. Then, to take full use of cluster resources, ReducePartition divides data into multiple partitions evenly. Next, taking into account the differences in computational capabilities among Executors, each task is assigned to Executor with the highest performance factor according to the greedy strategy. Finally, the results of the related algorithms and ReducePartition are compared by using WordCount benchmark and Sort benchmarks on heterogeneous Spark standalone cluster. The performance of the ReducePartition under different degree of data skew and different data size is analyzed. Experimental results show that the proposed algorithm can effectively reduce the impact of data skew on the total makespan of Spark big data applications, and the average total makespan is reduced by 30% to 50% while resource utilization is increased by 20%-30% on average.

KEYWORDS

data skew, load balancing, makespan, Spark

1 | INTRODUCTION

With Internet finance, traditional telecom services, public utilities, energy, transportation, and other industries rapidly enter the era of big data, leading to an exponential growing of data. In the face of this phenomenon, a large number of parallel computing frameworks for data processing have emerged. Hadoop,[1] as an open-source version for MapReduce, is a widely used distributed computing framework. Although the traditional MapReduce framework has achieved a great success in big data processing, it cannot deal with two types of problems that cannot be effectively handled in the current distributed computing framework: iterative computation and interactive computation. For the shortcomings of Hadoop, a new parallel computing framework, Spark, has emerged. Spark makes up the deficiency of traditional MapReduce framework especially in iterative computation.

Spark[2] is a new generation of distributed processing framework for big data following Hadoop. It has been rapidly pursued by academia and industry with its advanced design concept. It not only efficiently processes a large amount of data from different applications and data sources but also greatly reduces the number of disk I/Os by caching intermediate data of applications in memory and using a more powerful and flexible task scheduling mechanism based on directed acyclic graph (DAG).[2] Because Spark implements the DAG execution engine, which can efficiently process data streams based on memory, it is 100 times faster in terms of memory-based operations and 10 times faster in hard disk-based operations than Hadoop Mapreduce according to the official test results.[3]

Spark optimizes the execution logic of the application program through the Resilient Distributed Datasets (RDD) and the pipeline flow calculation method.[4] In Spark, each action calculation corresponds to a job. A job will be generated only when an action operator is invoked by an application. After a job is submitted to DAGScheduler, it will be split into groups of task, and each group of tasks is called a Stage. In terms of the dependency relation of RDD (narrow dependency and wide dependency), a Spark job can be divided into map stage, which contains

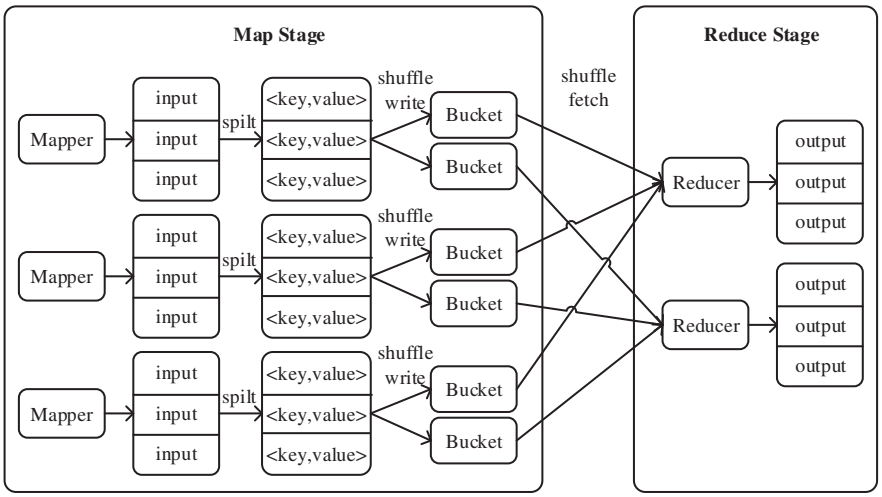

FIGURE 1 Map stage and reduce stage in Spark

the intermediate calculation results and reduce stage without any subsequent dependencies. The former stage is to generate shuffle files with intermediate results, and wait for the subsequent Reduce process to pull these files for the next round of calculations. Figure 1 displays the map stage and reduce stage in Spark. This parallel computing model enables Spark to efficiently process big data tasks through clustering.

Compared with other big data processing frameworks,[5] Spark has many advantages. However, the data skew problem due to the uneven distribution of input data and the unbalanced allocation of default partitioning algorithm in Spark will impact its performance. When data skew occurs, the execution time of some tasks will be significantly longer than other tasks especially in the reduce stage. Since tasks on the same stage have synchronization barriers, only when the slowest task in the current stage is completed can the execution of tasks in the subsequent stage begin. Therefore, when data skew occurs, the total makespan of the entire job will increase with reducing the program execution efficiency and further increasing the total energy consumption of the system.

To prevent straggler tasks in the existing Spark platform, a speculative execution mechanism similar to Hadoop MapReduce is adopted. To maximize system performance, ideally, the execution time of each task should be approximately the same. When the execution time of tasks varies greatly due to the uneven distribution of the original input data and other reasons, Spark detects that some tasks are straggler tasks and will try to start the same tasks with the same input data at the rest of the idle nodes, taking the calculation results of the tasks completed first as the final results. If some tasks of the cluster run slowly due to hardware aging, software misconfiguration, Spark's speculative execution mechanism can effectively reduce the total completion time of operations. However, Spark's speculative execution mechanism is powerless when the data skew is caused by uneven distribution of its input data. The reason is that because reexecution of tasks with the same input data on different machines will result in the same execution time, while redundant tasks occupy part of the resources of the cluster, resulting in an increase in the makespan of the entire job. Therefore, it is of great significance to deal with the problem of data skew in Spark to optimize the performance of applications and reduce the total makespan of applications.

Due to the uneven distribution of input data and the uneven distribution of default partitioning algorithm of Spark, data skew may occur, resulting in an increase in the makespan of big data applications. Thus, in this paper, we propose a balanced data allocation algorithm for data skew problem at reduce stage. We summarize the contribution of this work as follows:

- We propose a data balanced allocation method called ReducePartition to deal with data skew problem at reduce stage in Spark platform.
- We give the related definition of Reduce data skew problem and establishes the mathematical model for it.
- We introduce the overall architecture design of the data balanced allocation algorithm ReducePartition, which contains several subalgorithms.
- We compare ReducePartition with HashPartition, RangePartition, and SCID in four situations. Experimental results show that our proposed algorithm achieves a better performance than the other three methods especially when data skew is serious.

The rest of this paper is organized as follows. Section 2 gives an introduction to related work. Section 3 provides a mathematical model of the reduce-oriented data skew problem. The design idea of ReducePartition and pseudocode of related algorithms are discussed in Section 4. In Section 5, we compare the performance of ReducePartition through experiments. Finally, conclusions are given in Section 6.

2 | RELATED WORK

Spark is a popular memory computing framework, which is also troubled by the data skew problem.[6] How to solve the data skew problem of parallel distributed framework in it is a key issue. Although there are many studies on data skew problem for Hadoop, few researches have been done for Spark.

The data skew problem at reduce stage can be divided into two categories:(1) skew caused by partition function and (2) skew caused by some costly records in reduce tasks.[7] Spark divides all records by a Hash partition function by default, which does not guarantee balanced allocation.

The execution time of reduce task depends not only on the number of allocated keys but also on the distribution and proportion of data in the buffer.[8] He. et al[9] provided a comparative study on data skew in Hadoop in terms of architectures, main features, core algorithms, performance metrics, and evaluation methods, and summarized a few challenging problems as future research trends.

In view of the characteristics of data skew happening at reduce stage, decisions are made by collecting intermediate data during job execution in some studies. Ibrahim et al[10] proposed a partition optimization method for reduce stage. This method collects distribution information of key in advance and then sorts them in descending order according to the ratio of key's data locality and fairness. The key with the greatest fairness is assigned to reduce nodes first. Gufler et al[11] proposed a method to expand the number of partitions by multiplying the number of partitions of intermediate data before the end of map stage and then assigned them to reduce nodes by using a greedy policy–based bin packing algorithm. To determine more accurately the distribution of input data, they propose a TopCluster sampling method to estimate the partition status based on the frequency of keys extracted from each map task. However, the bin packing algorithm they use cannot provide good support for Sort applications. Liu et al[12] proposed a framework for mitigating partition skew in runtime by controlling the amount of resources allocated to each reduce task, which is different from the general method of redistributing workloads among reduce tasks and eliminates the overhead of repartition. Chen et al[13] proposed a lightweight algorithm called LIBRA to solve the problem of data skew among reducer tasks in MapReduce applications. LIBRA does not need to run any task in advance to sample the input data. Instead, it selects part of the original map task to implement the sampling logic and achieves a highly accurate estimation of the distribution of intermediate data by sampling a small part of the intermediate data. Moreover, it considers the heterogeneity of computing resources when balancing the load among reduce tasks. LIBRA has a high degree of parallelism with small overhead, but its application scope is limited.

Tang et al[6] proposed a splitting and combination algorithm called SCID to solve intermediate data skew in Spark 1.1, which makes different reduce tasks load balanced. Since the number of keys of input data cannot be calculated before data is processed by map tasks, they propose a reservoir sampling algorithm to detect the distribution of the keys of intermediate data. Compared with the original data loading mechanism, it sorts key-value tuples of each map task according to their size and then fills the data into relevant buckets in order. Once current tuples exceed the remaining capacity of current bucket, it will be split and the remaining data will be added to the next iteration. The total amount of data in each bucket is approximately equal through this process, and each reduce task obtains intermediate results from a specific bucket. However, this method is only effective for homogeneous clusters. In 2018, Tang et al[14] put forward a key reassigning and splitting partition algorithm to alleviate data skewness among reduce tasks. They use step-based rejection algorithm to sample input data. Besides, for different types of applications, they design two partition strategies: hash-based key reassigning algorithm and range-based key splitting algorithm. Liu et al[15] proposed a partition method called SP-Partition to deal with Spark streaming data skew problem in shuffle phase. In this strategy, they treat output data of map tasks as candidate samples and use systematic sampling to choose samples to predict the distribution of intermediate data. Cao et al[16] proposed a method to handle binary theta-joins on Spark. To reduce the computing cost, they filtered the join matrix before the map stage. In the process of data distribution, the partition algorithm they used is based on the ideas of range-based, M-theta-bucket and MDRP method, where MDRP is responsible for the load balance of each reducer.

Compared with the traditional cascaded two-way join algorithm, Afrati and Ullman[17] implemented an algorithm that converts multiway join into a single MapReduce job. The algorithm optimizes multiway join by minimizing duplicate tuples of input data and can detect whether an attribute is incorrectly contained in the keys of map side and then fix it. Hassan et al[18] proposed a new frequency adaptive algorithm called MRFA-Join for join operations of large-scale data sets based on a MapReduce programming model. To solve the problem of buffering all records of internal and external relations, Blans et al[19] proposed a semi-join algorithm for log processing and improve the sorting and merging operations in MapReduce. However, these algorithms still have serious performance defects in the face of data skew.

There are also some studies on data skew problem in distributed databases.[20-22] Ramakrishnan et al[23] proposed a static load balancing algorithm based on Oracle Loader for Hadoop to distribute reduce tasks. It distributes workloads through a load distributor, divides the large workload into small workloads, and uses a load assembler to distribute the medium workload to the reduce task to minimize the load of reduce side. However, these methods for load balancing of the distributed database cannot be directly used in Spark. Since tuples with the same attributes can be assigned to different groups for processing in the database, but keys with the same hash value must be executed on the same reduce task.

3 | PRELIMINARY KNOWLEDGE

To facilitate data balance partition algorithm in heterogeneous Spark clusters, some related definitions and problem model are presented in this section. The mathematical model of the data balanced distribution algorithm is established and the optimization objectives are proposed to clarify the problems to be solved.

3.1 | Definitions

Definition 1. CPU utilization of executor CU_i, which can be obtained by successively sampling the system values twice. The formula is as follows:

$$CU_i = \frac{total\_process\_2 - total\_process\_1}{total\_cpu\_2 - total\_cpu\_1} \times 100\%, \tag{1}$$

where *total_cpu_1* and *total_cpu_2* indicate the total CPU consumption time obtained by the first sampling and the second sampling, respectively. *total_cpu_*1 and *total_cpu_2* indicate the CPU time consumed by the executor process obtained by the first sampling and the second sampling, respectively.

Definition 2. Memory utilization of executor MU_i, which can be evaluated by Formula 2 is as follows:

$$MU_i = \left(1 - \frac{\text{remainingMem}}{\text{totalMem}}\right) \times 100\%, \tag{2}$$

where totalMem is the memory size configured for each executor, and remainingMem is the remaining memory size of each executor.

Definition 3. Weight of each Executor W_i, which indicates the free computing capacity of each executor. The executor with a higher value has the priority of processing data. This value is dynamically adjusted by workload, CPU utilization, and memory utilization of the executor. The initial value is calculated by Formula 3.

$$W_i = \text{Speed}_{\text{cpu}} \times (1 - R_{\text{cpu}}) \times (1 - R_{\text{mem}}), \tag{3}$$

where $\text{Speed}_{\text{cpu}}$ is the CPU frequency of the executor, and R_{cpu} and R_{mem} indicate the CPU utilization and memory utilization of executor, respectively. From Formula 3, we can see that the initial value of W_i is related to the hardware configuration and resource utilization of a node. The greater the node's CPU frequency, the lower the CPU utilization, and the lower the memory utilization, the greater the free computing capacity of the executor.

Definition 4. Performance factor of an executor f_i, which can be calculated by Formula (4) and Formula (5).

$$f_i = \frac{W_i}{f_{\text{avg}}}, \tag{4}$$

$$f_{\text{avg}} = \frac{\sum_{i=1}^{numOfExecutor} W_i}{numOfExecutor}, \tag{5}$$

where W_i represents the current weight of an executor, *numOfExecutor* is the number of total executors,and f_{avg} is the average of total weights of all Executor. The higher the performance factor f_i, the better the performance of executor, and the higher the computational efficiency.

3.2 | Problem modeling

In Spark, a job will only be generated when an application invokes an action, and each job can be divided into multiple stages based on the dependencies of RDD. Intermediate data between the upstream map stage and the downstream reduce stage can be represented as a collection of key-value pairs tuples = $\{(K_1,C_1),(K_2,C_2), \ldots ,(K_m,C_m)\}$. K_i represents the different key contained by the upstream map stage, and C_i represents the number of records corresponding to each key.

Let *F*, denoted as Formula 6, be the partition function corresponding to the intermediate data of the upstream stage to the downstream stage. *K* is the set of keys of intermediate data, and *n* is the number of partitions. That is, the intermediate data needs to be divided into *n* partitions.

$$F : K \to \{1, 2, \ldots, n\} \tag{6}$$

In Spark, the partition function specifies the key-value pairs contained in each partition. Formula 7 represents the number of records contained in each partition. p_j represents the *j*th partition.

$$p_j = \bigcup_{k \epsilon K : F(k)=j} C(k), 1 \leqslant j \leqslant n \tag{7}$$

To measure the load balancing degree of homogeneous clusters, we define Formula 8, and p_{avg} is the average number of records for each partition calculated by Formula 9.

$$\delta_1 = \sqrt{\frac{\sum_{j=1}^{n}(p_j - p_{\text{avg}})^2}{n}} \tag{8}$$

$$p_{\text{avg}} = \frac{\sum_{j=1}^{n} p_j}{n} \tag{9}$$

For homogeneous clusters, we want to minimize the number of key-value pairs for the maximum partition. Formula 10 shows our optimization objective.

$$\text{Minimize}_{j=1,2,\ldots,n} \max p_j \tag{10}$$

For heterogeneous clusters, we expect that the execution time of each task in reduce stage is roughly equal, that is, the standard deviation of the execution time among tasks is close to zero, which is calculated by Formula (11). $T(p_j)$ represents the execution time for each partition. However, the execution time of each partition cannot be obtained in advance before the corresponding task is completed. To reduce the computational complexity, the optimization objective of load balancing in heterogeneous clusters is modified as Formula (12), which is to minimize the number of records for the largest partition with consideration of the performance factor of executor.

$$\delta_2 = \sqrt{\frac{\sum_{j=1}^{n}\left(\frac{\sum_{j=1}^{n} T(p_j)}{n} - T(p_j)\right)^2}{n}} \tag{11}$$

$$\text{Minimize}_{j=1,2,\ldots,n} \max \frac{p_j}{f_i}, 1 \leqslant i \leqslant numOfExecutor \tag{12}$$

4 | DATA BALANCED PARTITION METHOD

4.1 | Method overview

The calculation skew of tasks in reduce stage is mainly caused by the imbalance of data allocation in upstream stage, that is, the default partition function does not consider the characteristics of uneven distribution of data and the difference of computing capacity of computing nodes when allocating data. For homogeneous clusters, it is only necessary to ensure the amount of data distributed between partitions is approximately the same. However, for heterogeneous cluster, differences in computing capacity between executors should be considered. Figure 2 presents the architecture of ReducePartition.

The whole process involves sampling algorithm, data balanced partition algorithm, weight adjustment algorithm, and task scheduling algorithm. Each of them will be discussed separately in the following parts. Here, for better understanding of our approach, we give an overview for ReducePartition:

1. Monitor the average CPU utilization and memory utilization of the compute nodes. After the executor process starts, initialize the weight of executor according to Formula (3).
2. Each computing node samples the local intermediate data according to the sampling rate, and then sends the local sampling information to the master node through message communication.
3. The master node collects the sampling information of all computing nodes and then predicts the overall characteristics of the data distribution.
4. Divide data into multiple partitions. The large key will be split by the partition number, which is an integer multiple of the total number of Executor cores.
5. Calculate the performance factor of executor according to Formula (4), and then assign the task to the Executor with the highest performance factor according to the greedy strategy.

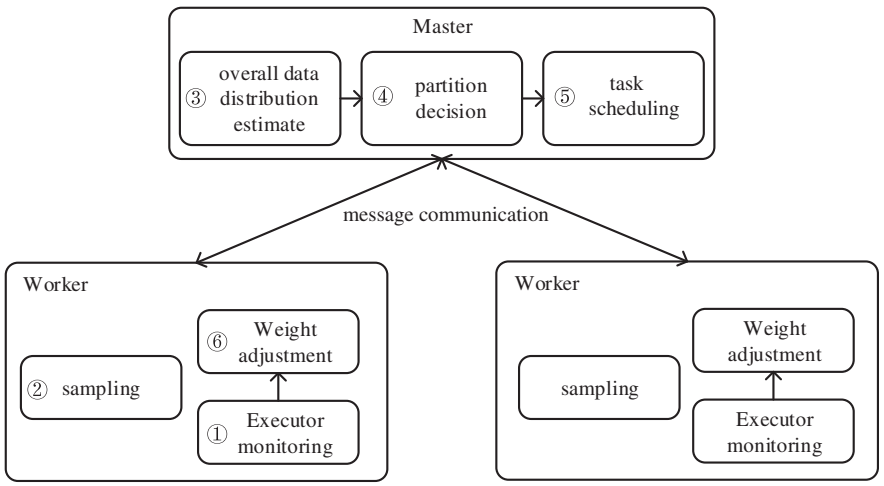

FIGURE 2 ReducePartition algorithm architecture

6. Executor's weight needs to be dynamically adjusted according to its workload and resource utilization in the whole process. Repeat step 5 until all the task is assigned.

4.2 | Sampling algorithm

To ascertain the distribution of intermediate data, sampling method is implemented, which can sample for different partitions with appropriate sampling size in parallel. In general, the sampling algorithm can be classified into two types: probability sampling and nonprobability sampling. In terms of probability sampling, each individual has a certain probability of being selected and the probability is greater than zero. However, for nonprobability sampling, the probability of each individual being selected is unknown. Besides, the sampling is not random.

An appropriate sampling algorithm can evaluate the overall distribution of data more effectively. Therefore, the selection of sampling algorithm needs to be based on specific scenes. Gulfer et al[24] drew support from histogram to deal with the load balancing problem in MapReduce framework. The algorithm TopCluster proposed by them first collects a local histogram Li of each partition maintained on mapper i. Next, each partition sends two information to controller: presence indicator for all local clusters and the histogram head, which is the largest local cluster. Finally, the controller estimates the global histogram by aggregating records from local histograms and computing clusters that are not in the local histograms. In their approach, two parameters are important. One is the cluster threshold and another is presence indicator. Although the way to set the threshold has been changed from static to dynamic, the presence indicator may cause false positive especially for the upper bound of a certain key. Besides, global histogram computed by TopCluster is just an approximation, we may get the wrong information of final intermediate data distribution even if TopCluster reduce error as much as possible. They take no consideration of the heterogeneity as well. As for Reservoir sampling, although it guarantees the randomness of sample, samples selected by it are not representative, which is not conducive to the evaluation of intermediate data since the process of it is totally random. To better estimate the distribution of intermediate data, the sampling algorithm we conducted in this paper chooses samples according to the proportion set in advance. In Section 2, we have mentioned the weakness of sampling algorithm in SCID and LIBRA methods, so the description will not be repeated here.

Because the nonprobabilistic sampling algorithm needs to import the subjective experience of researchers and the accuracy and error rate of the nonprobabilistic sampling algorithm cannot be evaluated; in addition, the input file data has great randomness in reality, and nonprobabilistic sampling algorithm is not suitable for our algorithm. Thus, we need to find an appropriate one from the probabilistic sampling algorithm. In the case of a large population of data, the sampling results of random sampling will produce large errors, systematic sampling takes long time for it involves sequential reading of files, and stratified sampling requires prior knowledge of the attributes of the data for stratification, which has poor generality. From these weaknesses, considering the speed and quality of sampling, we choose clustering sampling since it can strike a good balance between the two indicators.

Sampling process displayed in Algorithm 1 is the first step in ReducePartition. The sampling algorithm performs separately for each partition of an RDD, which accords with the idea of clustering sampling and it is performed at the local node. After sampling, the results of the local sample aggregation are sent to the Master node. For this reason, it is very fast since the network transmission is very small. The sampling algorithm can be summarized in four steps:

1. Calculate the total number of records *rddSize* for the input data through RDD operator. That is, each compute node calculates the total number of records of the partitions it owns, and finally transmits them to the Driver.
2. Calculate the total sample size *sampleSize* according to the preset sampling rate and then calculate the sample size for each partition *sampleSizePerPartition* according to the partition number of RDD. The higher the sampling rate is, the more accurate the data distribution is estimated, but the longer it takes.
3. Each compute node randomly samples data partition according to *sampleSizePerPartition*, counts the number of records of the local sample, and then sends (K_i, C_i) to the master node through message communication. K_i represents the key of record and C_i represents the number of records of the corresponding key.
4. The Master node summarizes the total number of samples from all compute nodes and estimates the overall distribution of data according to the preset sampling rate. That is, tuples={(K_1, C_1), (K_2, C_2), … , (K_m,C_m)} is obtained.

Algorithm 1 The sampling algorithm in reducepartition

Input: data *RDD*, partition number of RDD *partitionNum*
sampling rate α

Output: $tuples = \{(K_1, C_1), (K_2, C_2), \ldots, (K_m, C_m)\}$

1: Calculate the total size of data *rddSize*
2: Calculate the total sample size of data $sampleSize = rddSize \times \alpha$
3: Calculate sample size of each partition $sampleSizePerPartition = math.ceil(sampleSize/partitionNum)$
4: Random sampling for each partition
5: Each local node aggregates samples and transmits them to Master node through message communication
6: Master node statistics all samples
7: Estimate *tuples* according to sampling rate α

4.3 | Data balanced partition algorithm

We can obtain the set of intermediate data key-value pairs $\{(K_1, C_1), (K_2,C_2), \ldots, (K_m,C_m)\}$ after sampling algorithm introduced in the last section. Now, we can proceed to the next step: Data partition. A common data partition algorithm is shown in Algorithm 2. At first, it sorts the immediate tuples in descending order basing on C_i. Then, the algorithm employs greedy strategy to distribute C_m, which is the maximum value of key in tuples to a partition containing the minimum key-value pairs. Moreover, repeat the step until the immediate data distribution is finished. Although the algorithm guarantees that the same key is assigned to the same partition, the amount of data allocated to each partition may vary greatly. For example, an application includes three kinds of keys (K_1, K_2, K_3), and we should distribute it into three partitions. Assume that, through sampling algorithm, we get the amount of K_1, K_2, K_3 is 2000, 200, and 100, respectively. If we use Algorithm 2, data corresponding to K_1, K_2, and K_3 are assigned to the first partition, the second partition, and the third partition successively, which leads to a serious imbalance. Even if the partition with the largest amount of data is allocated to the node with the highest computational efficiency during the subsequent task allocation, the impact of serious data imbalance cannot be canceled out. Therefore, the execution time of tasks cannot be efficiently reduced.

Algorithm 2 Common data partition algorithm

Input: $tuples = \{(K_1, C_1), (K_2, C_2), \ldots, (K_m, C_m)\}$ data *rdd*, partition number of RDD *partitionNum*

Output: the partition to which Key belongs (*K_i*, p), $1 \le i \le m$, $1 \le p \le partitionNum$

```
Sort tuples in descending order according to C_i
for (K_i, C_i) in tuples do
    Assign (K_i, C_i) to the first partition
    Sort the partition in ascending order according to the total number of C_i
end for
```

It is obvious that the data size of each partition should be roughly the same to reduce the total execution time of tasks in the reduce stage. Moreover, there is a high chance that the amount of data allocated to each partition cannot be balanced under the constraint that the same key must be allocated to the same partition. This can be achieved by performing some conversion operations on the larger key. The complete data balanced partition algorithm is shown in Algorithm 3. The time complexity of this algorithm is $O(partitionNum \times log_2 partitionNum)$, where *partitionNum* represents the number of data partitions.

Algorithm 3 Data balanced partition algorithm

Input: $tuples = \{(K_1, C_1), (K_2, C_2), \ldots, (K_m, C_m)\}$

partition number *partitionNum*

Output: partition ID of data

```
Sort tuples in descending order according to C_m
Calculate p_avg according to formula (9)
Set maximum amount of data to be allocated for each partition to p_avg
for (K_i, C_i) in tuples do
    for j from 1 to partitionNum do
        if partition[j].residualCapacity ⩾ C_i then
            partition[j].residualCapacity −= C_i
            Record the placement that K_i will be assigned to the j-th partition
            break
        else
            partition[j].residualCapacity = 0
            Mark K_i as K_i_j, and record the placement that K_i will be assigned to the jth partition with partition[j].residualCapacity number of records
        end if
    end for
    Sort the partitions in descending order according to the remaining capacity
end for
(K_i, C_i) → (K_i_1, C_i_1),(K_i_2, C_i_2),...,(K_i_p, C_i_p) if necessary
if currentKey ϵ tuples.keys then
    Assign currentKey to the corresponding partition according to the recorded placement
else
    Assign currentKey to a partition using the default hash algorithm
end if
```

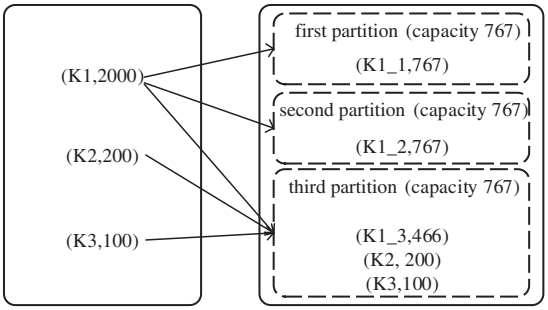

FIGURE 3 Balanced data partition in ReducePartition

As illustrated in Algorithm 3, the process of data balanced algorithm can be briefly divided into four steps:

1. Calculate the average number of records assigned to each partition p_{avg} according to estimated intermediate data tuples and Formula (9), and set the maximum amount of data to be allocated for each partition to p_{avg}.
2. Sort tuples in descending order of C_i and assign K_i to the first partition with the largest remaining capacity. If the remaining capacity of the partition is greater than or equal to C_i, assign K_i directly to the partition and update the remaining capacity of the partition. If the remaining capacity of the partition is smaller than C_i, allocate the remaining capacity of data to the partition, mark the allocated K_i as K_{i_1} and record the placement of K_{i_1}. Then, allocate the remaining K_i to the partition with the largest remaining capacity continually. If the size of the remaining capacity of the partition to be allocated next time is still not enough, mark the allocated key as $K_{i\_1}, K_{i\_2}, \ldots, K_{i\_p}$, successively, $1 \leqslant p \leqslant n$, and record the placement of them until K_i is allocated completely.
3. Repeat step 2 until all keys complete the prepartition process. If there is a conversion relationship from K_i to $K_{i\_1}, K_{i\_2}, \ldots, K_{i\_p}$, (K_i, C_i) needs to be converted to $(K_{i\_1}, C_{i\_1}), (K_{i\_2}, C_{i\_2}), \ldots, (K_{i\_p}, C_{i\_p})$, that is, the original K_i will be spilt into multiple new keys K_{i_j}, $1 \leqslant j \leqslant p$.
4. If the current key is in the sample set, then assign the key to the corresponding partition according to the placement information of K_i or K_{i_j} recorded in step 2. Otherwise, use the default hash algorithm to assign the key to the corresponding partition.

Notice that after the reduce stage is completed, when there is no semantic conflict in the context, the final calculation result needs to be merged according to the recorded partition attribution if there is a conversion relationship from K_i to $K_{i\_1}, K_{i\_2}, \ldots, K_{i\_p}$ when a job finishes. The process is similar to the partition.

After data partition operation, the amount of data of each partition is approximately the same. However, If the number of partitions follows the number of partitions of the last RDD of upstream stage, and the number of partitions is less than the number of preallocated CPU cores of current application, the resources of the cluster may not be maximized. For example, the number of preassigned CPU cores in the current application is 30, and the number of partitions in the last RDD of upstream stage is 20. Since each partition is processed by a CPU core in Spark, there are 10 core CPUs in an idle state. The allocated resources cannot be fully utilized, so the total makespan of current application must be extended.

Thus, to maximize preallocated resources of applications, we adjust the number of partitions and set the number of partitions to be an integer multiple of the number of preallocated CPU cores, namely, $partitionNum = \lambda \times \text{Core}_{app}$, where Core_{app} is the number of preallocated CPU cores for applications, $\lambda \geqslant 1$, which can be set by users. After this method is adopted, additional overhead is introduced in subsequent task scheduling. However, tests show that the overhead introduced in the subsequent task scheduling is negligible, and the performance improvement brought by maximizing the resource utilization of an application is completely greater than the overhead introduced in this part.

Here we take a WordCount application as an example to describe how to perform data balanced partition method. Suppose there are three kinds of key(K_1, K_2, and K_3) and they need to be divided into three partitions. The number of K_1, K_2, and K_3 estimated by the sampling algorithm is 2000, 200, and 100, respectively. The maximum amount of data to be allocated to the three partitions is set to $\left\lceil \frac{2000+200+100}{3} \right\rceil = 767$. First, assign K_1 to the first partition. Since the remaining capacity of the first partition 767 is less than the number of K_1 2000, allocate 767 keys to the first partition, and mark these keys as K_{1_1}. Since the remaining capacity of the second partition 767 is less than the remaining number of K_1 1233, allocate 767 keys to the second partition, and mark these keys as K_{1_2}. Since the remaining capacity of the third partition 767 is greater than the remaining number of K_1 466, all the remaining K_1 are allocated to the third partition. These keys are marked as K_{1_3}, and K_1 is allocated completely. The remaining capacity of the third partition is 301. Likewise, K_2 and K_3 are assigned to the third partition. Finally, the number of assigned records on the three partitions is 767, 767, and 766. The number of records among partitions is roughly the same, and the partitioning process of this example is shown in Figure 3.

4.4 | Weight adjustment algorithm

4.4.1 | Resource monitoring

The resource monitoring module contains two parts: Monitoring CPU and memory utilization of an executor. During the execution of Spark applications, due to network fluctuation, sudden stop or start of tasks and other reasons, the collected CPU utilization has certain errors.

To reduce the error, we introduce a negative feedback regulation mechanism in the control system theory, which adds the adjustment amount of the previous time at the current time. Formula 13 shows how to calculate the current monitoring value.

$$X(t) = X(t-1) + \sqrt[3]{\frac{X'(t)^2}{A} - X(t-1)} \tag{13}$$

$X(t\text{-}1)$ is the calculated value at last monitoring time. $X'(t)$ is the current monitoring value. A is used to control the magnitude of the adjustment amplitude, which is usually set to a value close to $X(t\text{-}1)$. Applying Formula 13 to monitor CPU utilization of executor, we can get the current CPU utilization of an executor $CU_i(t_j)$ by Formula 14.

$$\begin{cases} CU_i(t_{j-1}) + \sqrt[3]{CU'_i(t_j) - CU_i(t_{j-1})}, j \geq 1 \\ CU'_i(t_j), j = 0 \end{cases} \tag{14}$$

That is, when the current time is 0, the current CPU utilization $CU_i(t_j)$ is the monitoring value $CU'_i(t_j)$; otherwise, $CU_i(t_j)$ is determined by CPU utilization monitored at last moment $CU_i(t_j)$ and the current monitoring value $CU'_i(t_j)$. We set A to $CU'_i(t_j)$, whose value at every moment can be got from the system. It is more convenient to obtain memory utilization comparing to CPU utilization. *BlockManagerMasterEndpoint* includes two hash table: *blockManagerIdByExecutor* and *blockManagerInfo*. *blockManagerIdByExecutor* keeps one-to-one correspondence between executor and *BlockManagerId* while *blockManagerInfo* retains memory usage of each *BlockManagerId*. According to the transfer relation between two hash tables, we can gain the remaining memory of each executor from *blockManagerInfo*. Besides, the allocated memory size is already determined when the application is submitted. Therefore, memory usage of an executor can be calculated by Formula 2.

4.4.2 | Weight adjustment algorithm design

In a heterogeneous cluster environment, the computing capacity of each executor is significantly different. To improve resource utilization, some executors need to obtain more partitions, so it is necessary to measure the free computing capacity of each executor, which dynamically changes during the execution of tasks so as to facilitate decision-making during subsequent task scheduling. We assign an initial weight for each Executor according to Formula (3). When the weight adjustment period is reached, the weights are adjusted according to Algorithm 4, whose time complexity is $O(1)$ because it consists of a series of conditional statements. As Algorithm 4 shows, we first initialize the initial weight for each executor according to Formula (3), the computational capability counter $Capability_i$ and monitoring counter $Count_i$ of each executor. Then, we calculate CPU utilization CU_i and memory utilization MU_i for each executor. If the two indices do not exceed their upper bound $CU_{\text{upperbound}}$ and $MU_{\text{upperbound}}$ respectively, $Capability_i$ and $Count_i$ are increased by one. Otherwise, only the monitoring counter is increased. Next, the weight of executor is officially adjusted when the monitoring counter $Count_i$ reaches the adjustment conditions. If $Capability_i$ is greater than $\alpha \times \mathrm{T}$, executor weight increases. If $Capability_i$ is less than $\beta \times \mathrm{T}$, executor weight reduces. α and β are regulatory factors, which are set by users according to the specific scenario. Finally, reset $Capability_i$ and $Count_i$ to 0 each time the weight adjustment is completed. Repeatedly calculate and adjust CU_i, MU_i, $Capability_i$, and $Count_i$.

A key issue in the weight adjustment algorithm is how to set the weight adjustment cycle. Since the weight of executor is dynamic, to obtain more accurate data, an appropriate adjustment period needs to be set. If the adjustment period is set too large, the weight of executor is updated slowly, and the weight obtained during task scheduling is lagging behind. If the adjustment period is set too small, frequent adjustments will affect the efficiency of executor. Through tests by Tian et al,[25] it has been found that the size of the adjustment period is related to the specific application, and computation-intensive applications can set a larger adjustment period, I/O intensive application can set a small adjustment period. In their study, the observation period has been set to 10 seconds. In our experiment, we also adopt this value directly as the default adjustment period in the algorithm.

4.5 | Task scheduling algorithm

In Spark, the component *TaskScheduler* is responsible for the scheduling. Task assignment of a single taskset is completed by method *TaskSchedulerImpl.resourceOfferSingleTaskSet()*. The method first randomizes all executors, and then uses a polling mechanism to allocate tasks to executors that meet the CPU capacity constraints. The native task scheduling algorithm does not take into account the differences in executor's computational capabilities, which can easily lead to computational skew. Therefore, this paper introduces executor's performance factor to reflect the computational efficiency of executor, and the weight of executor is dynamically adjusted according to the weight adjustment algorithm. The weight adjustment algorithm and the task scheduling algorithm are performed asynchronously. The latest weight information of executor is obtained during each round of task scheduling. After data balanced partition algorithm, data in each partition is roughly the same. Besides, each partition corresponds to one task. To reduce the complexity of the algorithm while satisfying the optimization objective as much as possible, the tasks are assigned to the designated executor according to the greedy strategy. First, we calculate f_i of each executor according to Formula 4 and Formula 5. Then, sort f_i in descending order. Next, go through all the tasks and distribute tasks to executor with maximum f_i. If the number

Algorithm 4 Weight adjustment algorithm

Input: Initial weight of Executor W_i
period of Weight adjustment T
CPU utilization upper bound $CU_{\text{upperbound}}$
memory utilization upper bound $MU_{\text{upperbound}}$
regulatory factor α and β

Output: Weight of Executor after adjustment W_i

Initialize computational capability of each Executor $Capability_i = 0$
Initialize the monitor count for each Executor $Count_i = 0$
Calculate CPU utilization of each Executor CU_i
Calculate Memory utilization of each Executor MU_i
if $CU_i < CU_{\text{upperbound}} \&\& MU_i < MU_{\text{upperbound}}$ **then**
    $Capability_i++$
    $Count_i++$
else
    $Count_i++$
end if
if $Count_i == T$ **then**
    if $Capability_i > \alpha \times T$ **then**
        W_i++
        $Capability_i = 0$
        $Count_i = 0$
        output W_i
    end if
    if $Capability_i < \beta \times T$ **then**
        W_i--
        $Capability_i = 0$
        $Count_i = 0$
        output W_i
    end if
end if

of available CPU cores for the executor is greater than the number of CPU cores required for each task (the default value is 1 core), update the executor's available CPU cores while assigning the task and the task will run with maximum data locality on the executor. The detail process of task scheduling algorithm is shown in Algorithm 5.

Algorithm 5 Task scheduling algorithm

Input: $W = \{W_1, W_2, \ldots, W_n\}$
$Executors = \{Executor_1, \ldots, Executor_{numOfExecutor}\}$
$TaskSet = \{Task_1, \ldots, Task_{partitionNum}\}$

Output: $(Task_j, Executor_i)$

Calculate performance factor of Executor according to formula (4) and formula (5) $F = \{f_1, f_2, \ldots, f_n\}$
Sort *Executors* in descending order of performance factor
for $Executor_i$ in *Executors* **do**
    for $Task_j$ in *TaskSet* **do**
        if $!launched_j \&\& Executor_i.availableCpus \geqslant CPUS\_PER\_TASK$ **then**
            if $Task_j.locality(Executor_i)$ is the maximum **then**
                $Executor_i.availableCpus -= CPUS\_PER\_TASK$
                $launched_j = True$
                output $(Task_j, Executor_i)$
            end if
        end if
    end for
end for

The task scheduling algorithm introduces the performance factor of executor into the original task scheduling algorithm. The extra time complexity is $O(n \times log_2 n)$, and the total time complexity of this algorithm is $O(n \times log_2 n + n \times partitionNum)$, where n is the number of executor and *partitionNum* is the number of partitions that is equal to the number of tasks.

5 | EXPERIMENTS

5.1 | Experiment setting

In the experiment, we set up seven virtual machines on two heterogeneous tower servers to build a Spark standalone cluster. One virtual machine serves as the master node of Spark, and the other six virtual machines serve as worker nodes. Two types of Worker nodes are configured, each with three worker nodes. The same type of worker node is deployed on the same tower server. The configuration information of the master node and the worker node is shown in Table 1. What is more, *spark.default.parallelism* is set to 60, which is twice the total CPU number of the worker nodes. Each executor is configured with 4 GB of memory and two cores of CPU, which ensures that the cluster resources can be completely allocated. Clearly, a total of 15 executors are launched in the test environment.

The test data are Zipf distribution composite data sets with customized data inclination. Zipf distribution is a common data distribution in big data applications, which has a significant imbalance[26] with the characteristic that a small amount of data is more frequent while the vast majority of data are less frequent. The probability density function of the Zipf distribution is shown in Formula (15), where x is the ranking of the current key in all keys, n is the number of categories of keys, and α is used to define the degree of data skew. The degree of data skew is 0.0, indicating that the data distribution is balanced and the number of each key is equal. Thus, the larger the value of α, the greater the degree of data skew, which indicates that the data distribution is more unbalanced.

$$f(x) = \frac{1}{x^{\alpha} \sum_{i=1}^{n} (1/i)^{\alpha}} \tag{15}$$

Loads used in the paper are: *WordCount* and *Sort*, which are displayed in Table. 2. The data size of each load is 10 GB, 15 GB, and 20 GB. We use *groupByKey* instead of *reduceByKey* when implementing *WordCount* since *reduceByKey* does a map-side aggregation optimization leading to the fact that each Executor receives a small amount of data, thus, less data skew. Moreover, *sortByKey* operator is used in implementation of *Sort*. In the test, we conduct several tests and use the average as the result.

The implementation of *WordCount* is changed from using *reduceByKey* operator to *groupByKey* operator, because *reduceByKey* operator does a map-side aggregation optimization, the amount of data passed to each Executor is small, and the skew is not obvious. The implementation of *Sort* uses the *sortByKey* operator. Table 3 shows the parameter settings of ReducePartition.

TABLE 1 Configuration of master and worker in Spark cluster

Node type	Hardware configuration	Software configuration
Spark Master	6-core CPU, 12G memory	CentOS 7.2 x64, JDK 1.7
	300G hard disk, 10 Gigabit Ethernet	Hadoop 2.6.3, Scala 2.10.6, Spark1.6.2
Spark Worker Type 1	6-core CPU, 12 GB memory	CentOS 7.2 x64, JDK 1.7
	300 GB hard disk, 10 Gigabit Ethernet	Hadoop 2.6.3, Scala 2.10.6, Spark1.6.2
Spark Worker Type 2	4-core CPU, 8 GB memory	CentOS 7.2 x64, JDK 1.7
	300 GB hard disk, 10 Gigabit Ethernet	Hadoop 2.6.3, Scala 2.10.6, Spark1.6.2

TABLE 2 Detail of workload

Workload	Test data	Data size
WordCount	Zipf distribution dataset with different degree of data skew	10 GB, 15 GB, 20 GB
Sort	Zipf distribution dataset with different degree of data skew	10 GB, 15 GB, 20 GB

TABLE 3 Parameter settings of ReducePartition

Parameter	Description	Value
partitionNum	partition number	60
$CU_{upperbound}$	CPU utilization upper bound	0.9
$MU_{upperbound}$	memory utilization upper bound	0.9
T	period of Weight adjustment	10s
α	regulatory factor	0.8
β	regulatory factor	0.2

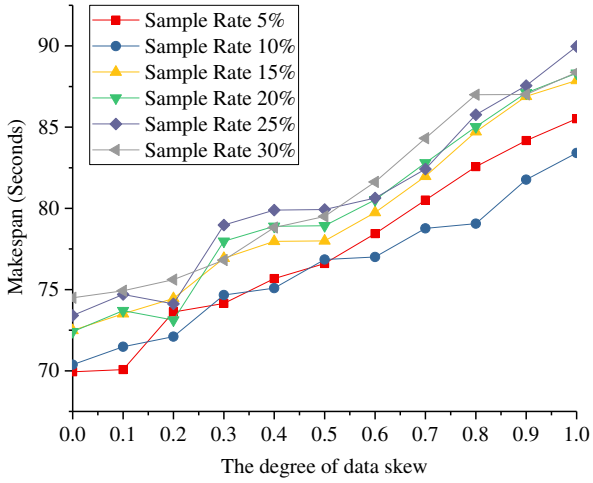

FIGURE 4 Total makespan of ReducePartition under different sampling rates in WordCount benchmark

In our experiments, we adopt makespan to be an evaluation index of algorithm performance and the following partition methods are chosen for comparison.

- HashPartition. It is the default partition method in the original MapReduce frameworks, such as Hadoop and Spark, which uses Java's Object.hashCodemethod to determine the partition as *partition = key.hashCode()%numPartitions*. Although this method is fast for intermediate data, it easily makes the data distribution unbalanced.
- RangePartition.[27] The partitioner mainly contains two steps. First, extract sample data from entire RDD, sort it and calculate the max Key of each partition, which forms Array *rangeBounds*. Second, judge the range where the key is located in the *rangeBounds*, and give the partition id of the key in the next RDD.
- SCID.[6] is a splitting and combination algorithm for intermediate skew data that makes the load of different reduce tasks is balanced. A sampling algorithm based on reservoir sampling is proposed to detect the distribution of the keys in intermediate data. SCID sorts the key/value tuples of each map task according to their sizes and fills them into the relevant buckets orderly. Once the data size exceeds the residual volume of the current bucket, it will be split and the remaining data will be added to the next iteration.

5.2 | WordCount benchmark

1. Under the same amount of data, the influence of different data sampling rates on total makespan of Reducepartition. Figure 4 compares the total makespan of ReducePartition under different sampling rates in the WordCount benchmark. The test data is 10-GB Zipf distributed synthetic data with the degree of data skew varying from 0.0 to 1.0. From Figure 4, it can be seen that under the same degree of data skew, the difference in sampling rate has less influence on the total makespan of ReducePartition, and the gap between the maximum total makespan and the minimum total makespan in each group of tests is within 10 seconds, indicating that it is efficient to adopt a sampling strategy in Spark. In the six sampling rates, as the degree of data skew increases, the total makespan of ReducePartition increases while the gradient is small. When the sampling rate is set to 10%, the total makespan of ReducePartition is at a minimum in multiple groups of tests, so the sampling rate is set to 10% in the subsequent WordCount benchmark test of ReducePartition.
2. Under the same amount of data, the influence of different data skew on total makespan of different algorithms. Figure 5 compares the total makespan of HashPartition, RangePartition, SCID, and ReducePartition under different degree of data skew in WordCount benchmark. The test data is 10-GB Zipf distributed synthetic data with the degree of data skew varying from 0.0 to 1.0. As the degree of data skew increases, the total makespan of all these four algorithms increases. The gradient of RangePartition is the largest, followed by the HashPartitioner, and the gradient of ReducePartition is the smallest. The total makespan of SCID is greater than ReducePartition in the 11 groups of tests and the gap between the two algorithms is larger and larger as the degree of data skew increases. There is a similar trend for RangePartition and HashPartition. When the data skew is small (eg, the degrees are 0.0 and 0.1), the makespan of ReducePartition is slightly larger than RangePartition and HashPartition for the overhead of sampling, data balanced partition and task scheduling is greater than the time reduced by ReducePartition. When the skew degree is beyond 0.3, ReducePartition shows its advantage and with the increase of data skew, the advantage of ReducePartition becomes more and more obvious. When the degree of data skew is 1.0, ReducePartition reduces the total makespan by 52.0% compared to HashPartition, 59.1% compared to RangePartition, and 32.2% compared to SCID.
3. Under the same amount of data, influence of different data inclination on average utilization of cluster resources of different algorithms. Figure 6 shows the comparison of the four algorithms in the average utilization of cluster resources under different degree of data skew. The test data is 10-GB Zipf distributed synthetic data with the degree of data skew varying from 0.0 to 1.0. As the degree of data skew increases, the average utilization of cluster resources for them both decreases. The gradient of RangePartition is the largest, followed by the HashPartitioner, and the gradient of SCID and ReducePartition is almost the same. In all 11 groups of tests, the average utilization of cluster

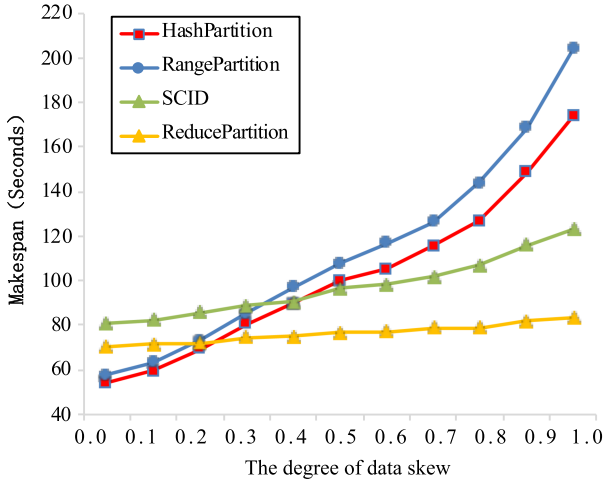

FIGURE 5 Total makespan of different algorithms under different degree of data skew in WordCount benchmark

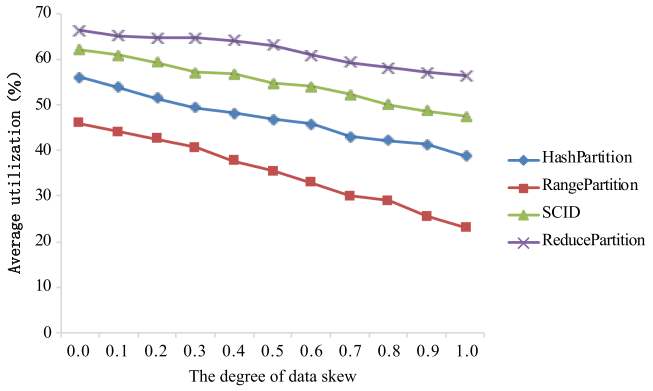

FIGURE 6 Average utilization of different algorithms under different degree of data skew in WordCount benchmark

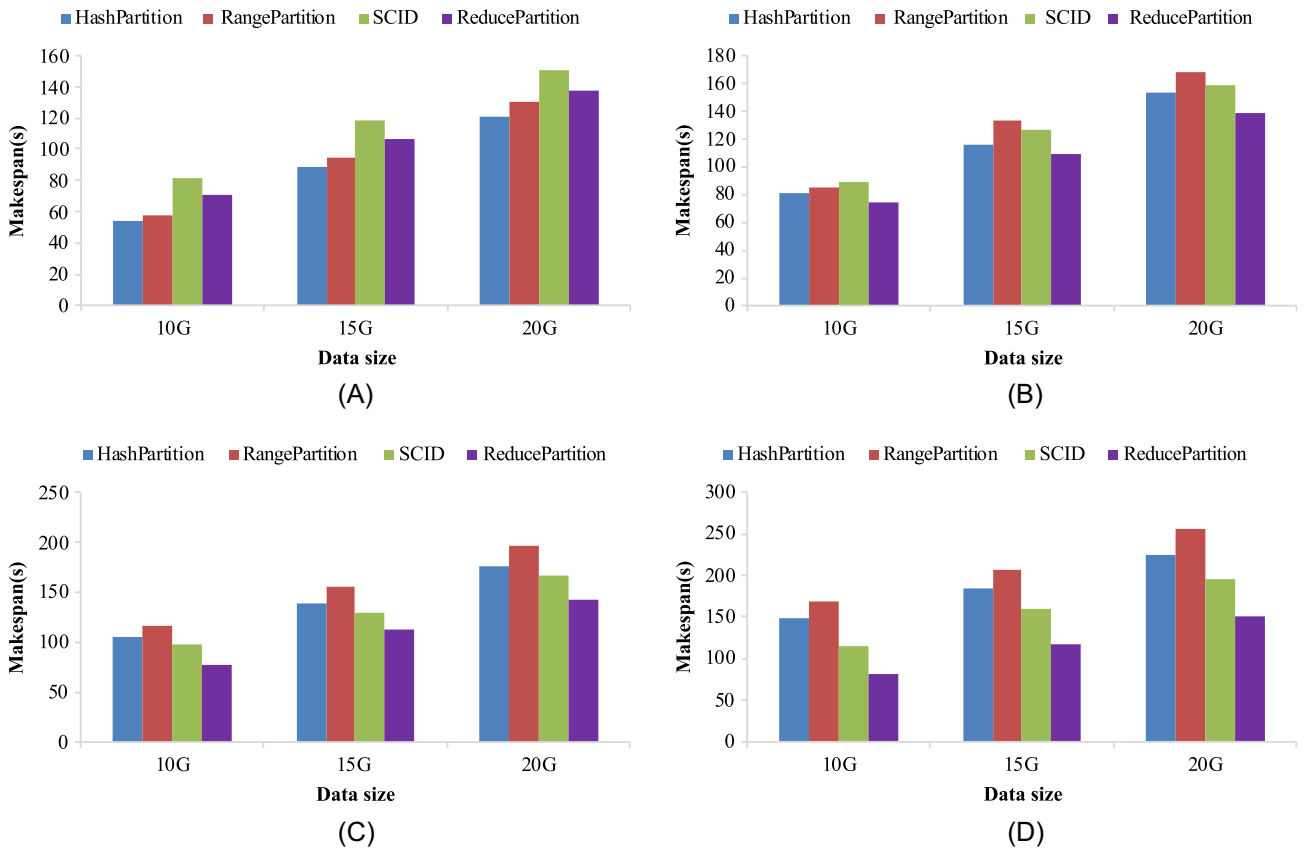

FIGURE 7 Total makespan of different algorithms under different data size in WordCount benchmark. A, skew = 0.0; B, skew = 0.3; C, skew = 0.6; D, skew = 0.9

resources of ReducePartition is always greater than the others, and the gap of the average utilization gradually increases as the degree of data skew increases. When the degree of data skew is 1.0, the average utilization of ReducePartition is 17.7% higher than that of HashPartition, 33.3% higher than RangePartition and 8% higher than SCID.

4. Under the same degree of data skew, the influence of different amount of data on makespan of different algorithms.
   Figure 7 compares the total makespan of HashPartition, RangePartition, SCID, and ReducePartition under different data size. The test data is Zipf distributed synthetic data of 10 GB, 15 GB, and 20 GB. Four groups of tests are performed with the degree of data skew is 0.0, 0.3, 0.6, and 0.9, respectively, and results are demonstrated in Figure 7A to Figure 7D. As the degree of data skew increases, the total makespan of HashPartition, RangePartition, SCID, and ReducePartition increases. As the degree of data skew increases, the makespan gap of the four algorithms increases. When the data is evenly distributed (the degree of data skew is 0.0), HashPartion performs best while ReducePartition ranks third. As the amount of data increases, the makespan differences between ReducePartition and HashPartition is smaller and smaller. However, when data distribution is unbalanced, ReducePartiiton is the champion of these four algorithms. Moreover, with the increase of degree of data skew and data size, the makespan of ReducePartition is growing more and more slowly. When the degree of data skew is 0.9 and data size is 20 GB, ReducePartition reduces the total makespan by 32.9% compared to HashPartition, 41.1% compared to RangePartition and 23.2% compared to SCID.

5.3 | Sort benchmark

We conduct a similar experiment for sort benchmark. The total makespan of ReducePartition under different sampling rates is shown in Figure 8. Under the same degree of data skew, the difference in sampling rate has less influence on the total makespan of ReducePartition. When the degree of data is 1.0, the gap between the maximum total makespan and the minimum total makespan is the largest, but it is still within an acceptable range, indicating that it is efficient to adopt a sampling strategy in Spark. In the six sampling rates, as the degree of data skew increases, the total makespan of ReducePartition increases but the gradient is small. When the sampling rate is set to 10%, the total makespan of ReducePartition is at a minimum in multiple groups of tests, so the sampling rate is set to 10% in the subsequent Sort benchmark test of ReducePartition.

Figure 9A compares the total makespan of HashPartition, RangePartition,[27] SCID, and ReducePartition under different degree of data skew. The test data is 10-GB Zipf distributed synthetic data with the degree of data skew varying from 0.0 to 1.0. As the degree of data skew increases, the total makespan of HashPartition, RangePartition, SCID, and ReducePartition increases. The gradient of RangePartition is the largest, followed by the HashPartitionerand SCID, and the gradient of ReducePartition is the smallest. When the degree of data skew is less than 0.6, the total makespan of RangePartition is less than that of HashPartition. When the degree of data skew is greater than 0.6, the total makespan of RangePartition is greater than that of HashPartition, and as the degree of data skew increases, the gap between the total makespan of RangePartition and HashPartition gradually increases. When the degree of data skew is small (data skew is between 0.0 and 0.1), the total makespan of ReducePartition is slightly larger than that of HashPartition because the overhead is greater than the time reduced by ReducePartition. When the degree of data skew is 0.3, the total makespan of ReducePartition is slightly higher than that of RangePartition, which is slightly lower than that of HashPartition. When the data distribution is more unbalanced (data skew is greater than or equal to 0.4), the total makespan of ReducePartition is the minimum, and as the degree of data skew increases, the time reduced by ReducePartition is increasing. When the degree of data skew is 1.0, ReducePartition reduces the total makespan by 33.7%, 43.8%, and 18.9% compared to HashPartition, RangePartition, and SCID.

Under different degree of data skew, the average utilization of cluster resources for HashPartition, RangePartition, SCID, and ReducePartition is shown in Figure 9B. As the degree of data skew increases, the average utilization of cluster resources for them both decreases. The gradient of ReducePartition is the smallest. In all 11 groups of tests, the average utilization of cluster resources of ReducePartition is always greater than

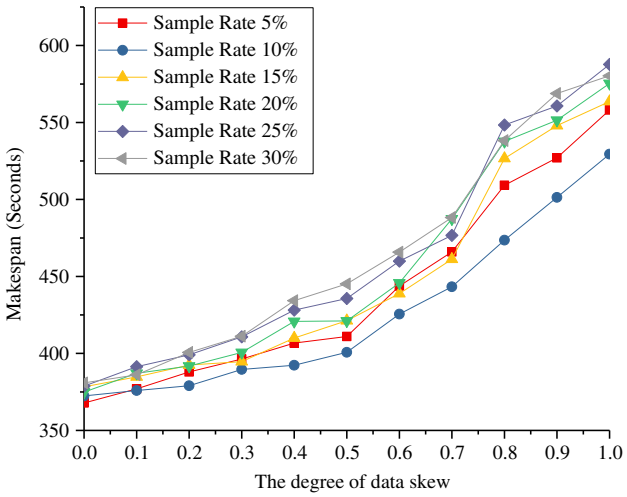

FIGURE 8 Total makespan of ReducePartition under different sampling rates in Sort benchmark

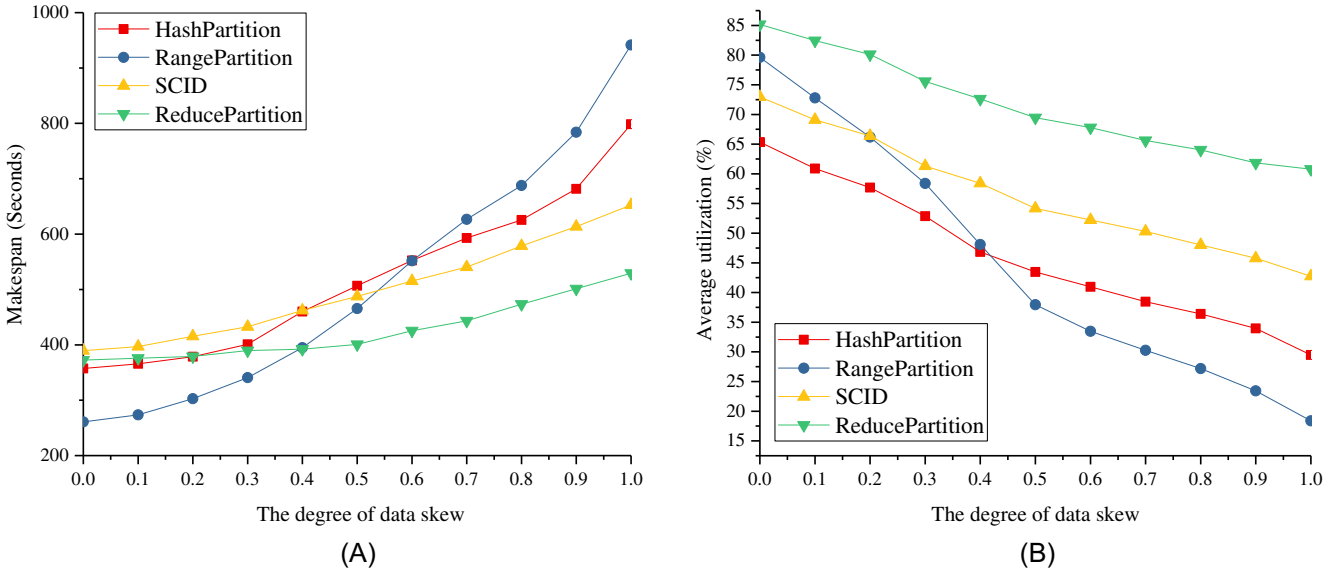

FIGURE 9 Total makespan and Average utilization of different algorithms under different degree of data skew in Sort benchmark. A, total makespan; B, average utilization

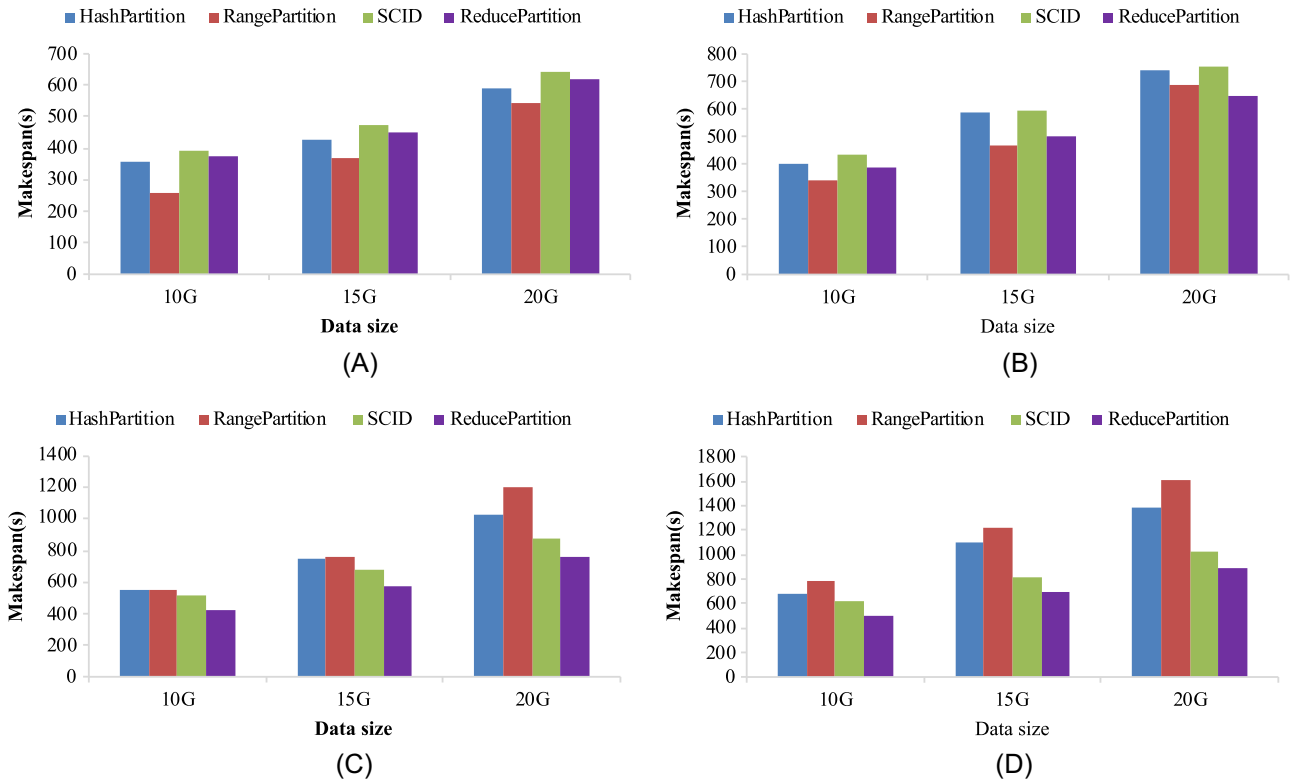

FIGURE 10 Total makespan of different algorithms under different data size in Sort benchmark. A, skew = 0.0; B, skew = 0.3; C, skew = 0.6; D, skew = 0.9

the others, and the gap of the average utilization gradually increases as the degree of data skew increases. When the degree of data skew is 1.0, the average utilization of ReducePartition is 31.3%, 42.4%, and 18.0% higher than that of HashPartition, RangePartition, and SCID.

Figure 10 compares the total makespan of HashPartition, RangePartition, SCID, and ReducePartition subjecting to the same degree of data skew and different data size. The test data is Zipf distributed synthetic data of 10 GB, 15 GB and 20 GB. We also perform four groups of tests shown in Figure 10A to Figure 10D, where the degree of data skew for each group is 0.0, 0.3, 0.6, and 0.9, respectively. As the degree of data skew increases, the total makespan of the four algorithms all increases. In the first two groups of tests, where the data skew is not that serious, RangePartition performs well while ReducePartition takes the second place. As the degree of data skew and the amount of data increase, ReducePartition performs better and better. Specifically, when the degree of data skew is 0.9 and the data size is 20 GB, ReducePartition reduces the total makespan by 36.1%, 45.1%, and 13.3% compared to HashPartition, RangePartition, and SCID.

5.4 | PageRank benchmark

PageRank is another benchmark which obeys Zipf distribution. As with the previous two benchmarks, the test data size is 10 GB, 15 GB, and 20 GB. Restricted by the experimental environment, the test environment has changed a little. We build four virtual machines on one tower server, where one virtual machine serves as the master node of Spark, and the other three virtual machines serve as worker nodes. In the experiment, two types of worker nodes were configured, and the CPU cores and memory capacity of each worker node were different. The configuration information for Master node and Worker node is shown in Table 4. IP distribution and running service statistics of each node in the cluster are displayed in Table 5.

Similar experiments have been conducted for PageRank benchmark. From Figure 11 to Figure 13 show the test results. We will analyze the test results in detail in the following parts. First, we tested the effect of sampling rate on total makespan. The data size is 10 GB whose skew is from 0.0 to 1.0. As shown in Figure 11, the sampling rate has little effect on the total completion time especially when there is no data skew. Even when the skew is 1.0, the difference between the maximum and minimum of makespan is also within the acceptable range. ReducePartition achieves the minimum makespan when sampling rate is set to 10%. Therefore, in subsequent tests, we set the sampling rate at 10%.

Figure 12A compares the total makespan of HashPartition, RangePartition,[27] SCID, and ReducePartition under different degree of data skew. The test data is 10-GB Zipf distributed synthetic data with the degree of data skew varying from 0.0 to 1.0. As the degree of data skew increases, the total makespan of HashPartition, RangePartition, SCID, and ReducePartition increases. According to the results, we know that RangePartition takes the longest time while ReducePartition shows advantage when the data is in serious unbalanced. When the degree of data skew is small (data skew is between 0.0 and 0.1), the total makespan of ReducePartition is slightly larger than that of HashPartition and SCID because the overhead of sampling procee is greater than the time reduced by ReducePartition. When the degree of data skew is 0.2, 0.3, and 0.4, HashPartition, SCID, and ReducePartition almost finish at the same time. When the data distribution is more unbalanced (data skew is greater than or equal to 0.5), the growth trend of ReducePartition is slowing down and its curve is at the bottom. In particular when the degree of data skew is 1.0, ReducePartition reduces the total makespan by 14.6%, 23%, and 2.8% compared to HashPartition, RangePartition, and SCID.

The average utilization of cluster resources for HashPartition, RangePartition, SCID and ReducePartition under different degree of data skew has been tested. The results are displayed in Figure 12B. As the degree of data skew increases, the average utilization of cluster resources

TABLE 4 Configuration of Master and Worker in Spark cluster

Node type	Hardware configuration	Software configuration
Spark Master	8-core CPU, 8 GB memory	Red Hat 4.8.5-36, JDK 1.7
	500 GB hard disk, 10 Gigabit Ethernet	Hadoop2.7.7, Scala 2.13.0, Spark 2.4.4
Spark Worker Type1	8-core CPU, 8 GB memory	Red Hat 4.8.5-36, JDK 1.7
	500 GB hard disk, 10 Gigabit Ethernet	Hadoop2.7.7, Scala 2.13.0, Spark 2.4.4
Spark Worker Type2	4-core CPU, 8 GB memory	Red Hat 4.8.5-36, JDK 1.7
	300 GB hard disk, 10 Gigabit Ethernet	Hadoop 2.7.7, Scala 2.13.0, Spark 2.4.4

TABLE 5 Cluster node IP partitioning and running service statistics

Node name	IP	Running service
Spark Master	10.8.8.17	NameNode
		SecondaryNameNode
		ResourceManager
		JobHistoryServer
		HistoryServer
		Master
SparkWorker1	10.8.8.97	DataNode
SparkWorker2	10.8.8.98	NodeManager
SparkWorker3	10.8.8.99	Worker

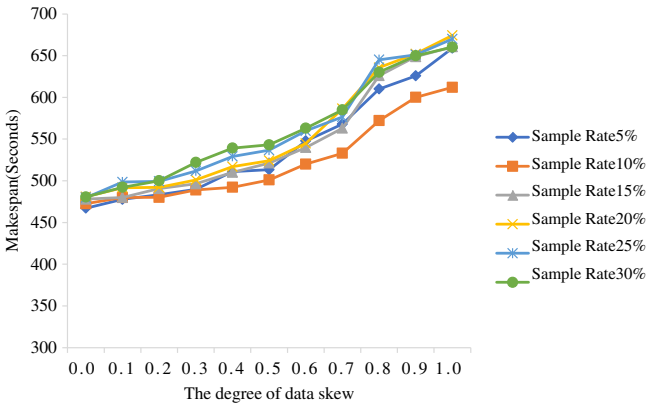

FIGURE 11 Total makespan of ReducePartition under different sampling rates in PageRank benchmark

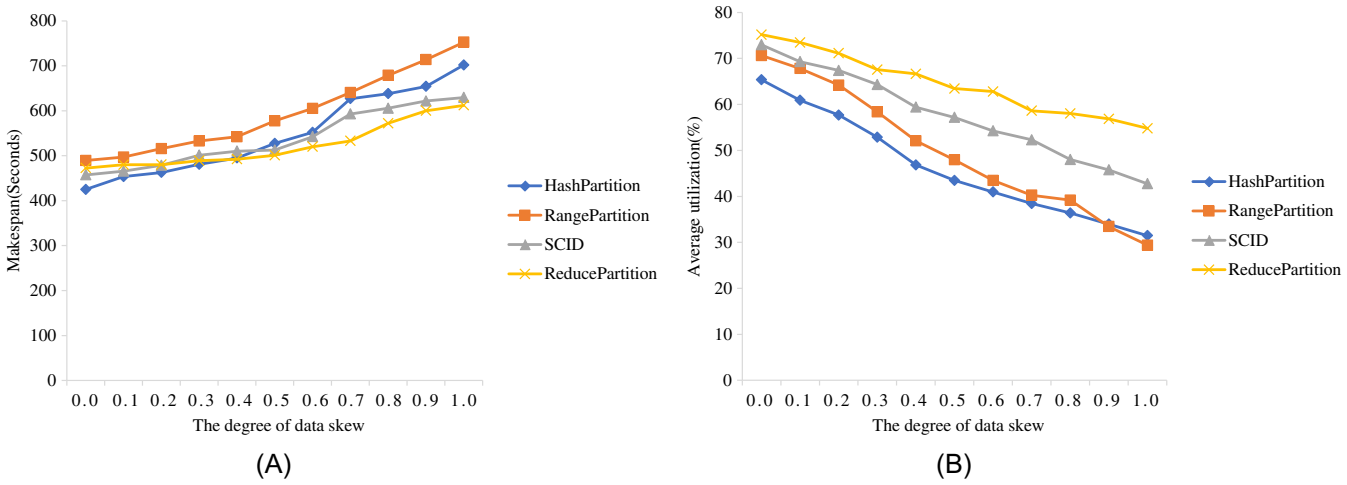

FIGURE 12 Total makespan and average utilization of different algorithms under different degree of data skew in Sort benchmark. A, total makespan; B, average utilization

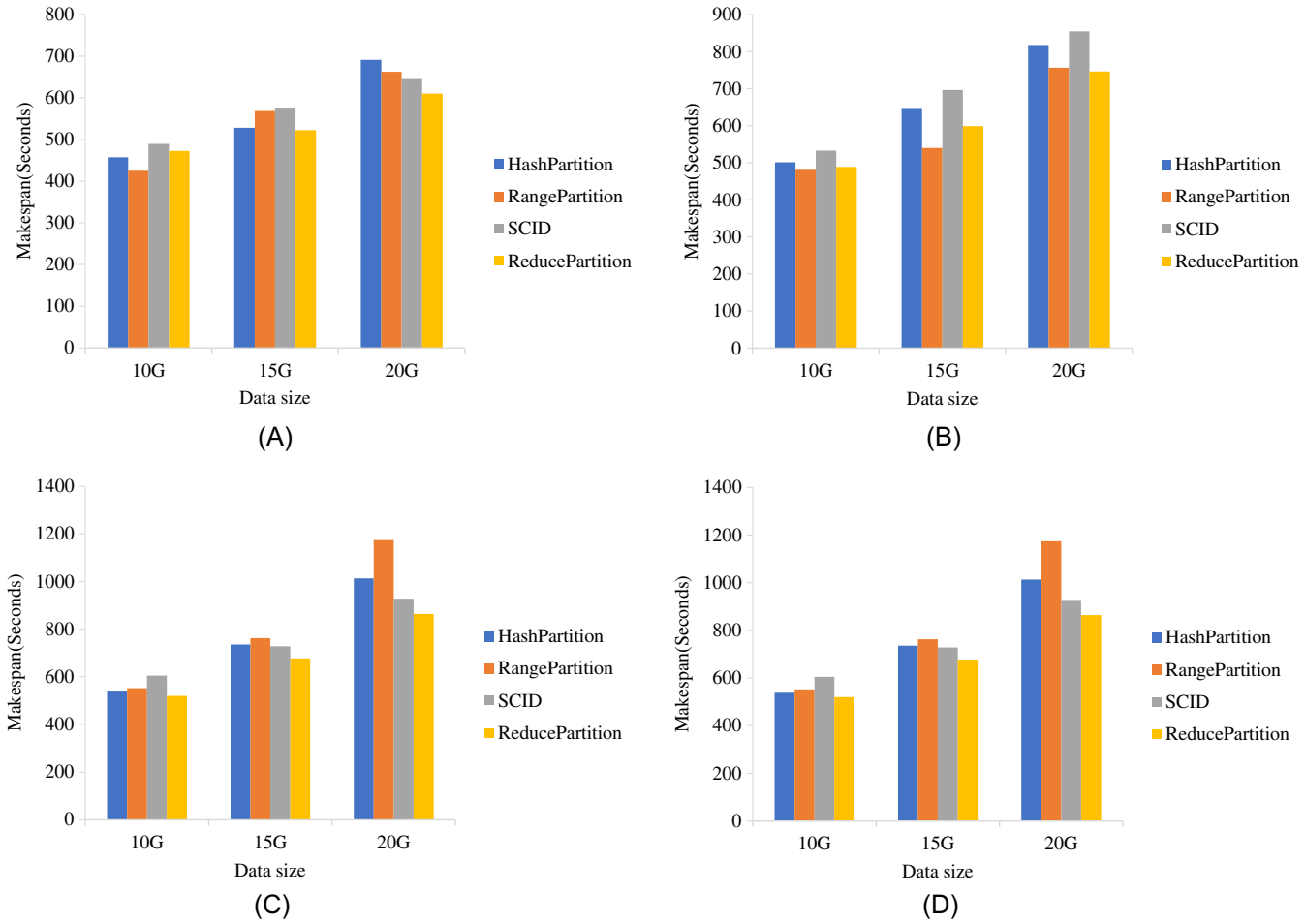

FIGURE 13 Total makespan of different algorithms under different data size in PageRank benchmark. A, skew = 0.0; B, skew = 0.3; C, skew = 0.6; D, skew = 0.9

of them all decreases. The gradient of ReducePartition is the smallest. In all 11 groups of tests, the average utilization of cluster resources of ReducePartition is always greater than the others, and the gap of the average utilization gradually increases as the degree of data skew increases. When the degree of data skew is 1.0, the average utilization of ReducePartition is 42.6%, 46.4%, and 22.0% higher than that of HashPartition, RangePartition, and SCID.

Figure 13 shows the comparisons of the total makespan of HashPartition, RangePartition, SCID, and ReducePartition subjecting to the same degree of data skew and different data size. Figure 13A to Figure 13D show the test results of four groups with 0.0, 0.3, 0.6, and 0.9 data skew

degrees, respectively, and each of them presents the situation with data of 10 GB, 15 GB, and 20 GB. As the degree of data skew and data size increase, the total makespan of the four algorithms all increases. The first two groups in which the data skew is not that serious, RangePartition performs well and ReducePartition just follows it. As the degree of data skew increases, no matter how much the data we test, ReducePartition always get the best results. Specifically, when the degree of data skew is 0.9 and the data size is 20 GB, ReducePartition reduces the total makespan by 21.6%, 28.0%, and 16.3% compared to HashPartition, RangePartition, and SCID.

6 | CONCLUSION

In the era of big data, the small quality improvement of big data platform can lead to greater execution efficiency. In this work, we propose a load balancing algorithm called ReducePartition to deal with data skew problem in Spark. The algorithm predicts the overall characteristics of data distribution through appropriate sampling algorithm. Then, to make full use of cluster resources, the data is evenly divided into multiple partitions. The number of partitions is an integer multiple of the total number of Executor cores, and the difference in computational capabilities among Executors is also considered. The task is assigned to the executor with the highest performance factor according to the greedy strategy. Experimental results show that the algorithm proposed in this paper can effectively reduce the total makespan of big data applications with small overhead, especially when the data is highly skewed. Besides, another issue we plan to investigate in the future is to propose efficient scheduling algorithms to deal with data skew in map stage.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under grants 61672136 and 61828202.

ORCID

Wenxia Guo https://orcid.org/0000-0002-0744-5745
Wenhong Tian https://orcid.org/0000-0002-5551-9796

REFERENCES

1. Dean J, Ghemawat S. MapReduce: simplified data processing on large clusters. *Commun ACM*. 2008;51(1):107-113.
2. Han Z, Zhang Y. Spark: a big data processing platform based on memory computing; Paper presented at: Seventh International Symposium on Parallel Architectures, Algorithms and Programming (PAAP); 2015; Nanjing, China.
3. Apache Software Foundation. Spark website. https://spark.apache.org/. Accessed 2016.
4. Zaharia M, Chowdhury M, Das T, et al. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. Paper presented at: 9th USENIX conference on Networked Systems Design and Implementation; 2012; San Jose, CA
5. Sfrent A, Pop F. Asymptotic scheduling for many task computing in big data platforms. *Inform Sci*. 2015;319:71-91.
6. Tang Z, Zhang X, Li K, Li K. An intermediate data placement algorithm for load balancing in spark computing environment. *Future Gen Comput Syst*. 2018;78:287-301.
7. Dhawalia P, Kailasam S, Janakiram D. Chisel++: handling partitioning skew in mapreduce framework using efficient range partitioning technique. Paper presented at: 6th International Workshop on Data Intensive Distributed Computing; 2014; Vancouver, Canada.
8. Cheng L, Kotoulas S. Efficient skew handling for outer joins in a cloud computing environment. *IEEE Trans Cloud Comput*. 2018;6(2):558-571.
9. He M, Li G, Huang C, Ye Y, Tian W. A comparative study of data skew in Hadoop. In: 2017 VI International Conference on Network, Communication and Computing; 2017; Kunming, China.
10. Ibrahim S, Jin H, Lu L, Wu S, He B, Qi L. LEEN: locality/fairness-aware key partitioning for MapReduce in the cloud. Paper presented at: 2010 IEEE Second International Conference on Cloud Computing Technology and Science; 2010.
11. Gufler B, Augsten N, Reiser A, Kemper A. Handling data skew in MapReduce. Paper presented at: 1st International Conference on Cloud Computing and Services Science; 2011; Noordwijkerhout, Netherlands.
12. Liu Z, Zhang Q, Ahmed R, Boutaba R, Liu Y, Gong Z. Dynamic resource allocation for MapReduce with partitioning skew. *IEEE Trans Comput*. 2016;65(11):3304-3317.
13. Chen Q, Yao J, Xiao Z. LIBRA: lightweight data skew mitigation in MapReduce. *IEEE Trans Parallel Distrib Syst*. 2015;26(9):2520-2533.
14. Tang Z, Lv W, Li K, Li K. An intermediate data partition algorithm for skew mitigation in Spark computing environment. *IEEE Trans Cloud Comput*. 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 Gen Comput Syst*. 2018;86:1054-1063.
16. Cao S, Haihong E, Song M, Zhang K. Optimization of data distribution strategy in theta-join process based on Spark. Paper presented at: 2nd International Conference on Algorithms, Computing and Systems; 2018; Beijing, China.
17. Afrati FN, Ullman JD. Optimizing multiway joins in a map-reduce environment. *IEEE Trans Knowledge Data Eng*. 2011;23(9):1282-1298.
18. Hassan MAH, Bamha M, Loulergue F. Handling data-skew effects in join operations using MapReduce. *Procedia Comput Sci*. 2014;29:145-158.
19. Blanas S, Patel JM, Ercegovac V, Rao J, Shekita EJ, Tian Y. A comparison of join algorithms for log processing in MapReduce. Paper presented at: 2010 ACM SIGMOD International Conference on Management of Data; 2010; Indianapolis, IN.
20. Phinney M, Lander S, Spencer M, Shyu C-R. Cartesian operations on distributed datasets using virtual partitioning. Paper presented at: 2016 IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService); 2016; Oxford, UK.

21. Xu Y, Kostamaa P. Efficient outer join data skew handling in parallel DBMS. *Proc VLDB Endowment*. 2009;2(2):1390-1396.
22. Walton CB, Dale AG, Jenevein RM. A taxonomy and performance model of data skew effects in parallel joins. Paper presented at: 17th International Conference on Very Large Databases; 1991; Barcelona, Spain.
23. Ramakrishnan SR, Swart G, Urmanov A. Balancing reducer skew in MapReduce workloads using progressive sampling. Paper presented at: 3rd ACM Symposium on Cloud Computing; 2012; San Jose, CA.
24. Gufler B, Augsten N, Reiser A, Kemper A. Load balancing in mapreduce based on scalable cardinality estimates. Paper presented at: 2012 IEEE 28th International Conference on Data Engineering; 2012; Washington, DC.
25. Wenhong T, Guozhong L, Yu C, Chaojie H, Wutong Y. Combined load balancing and energy efficiency in Hadoop. *J Tsinghua Univ Sci Technol*. 56(11):1226-1231.
26. Zhang Y, Nadarajah S. Flexible heavy tailed distributions for big data. *Annal Data Sci*. 2017;4(3):421-432.
27. Rangepartitioner. https://spark.apache.org/docs/2.3.0/api/java/org/apache/spark/RangePartitioner.html. Accessed 2016.

How to cite this article: Guo W, Huang C, Tian W. Handling data skew at reduce stage in Spark by ReducePartition. *Concurrency Computat Pract Exper*. 2020;32:e5637. https://doi.org/10.1002/cpe.5637