Journal of Grid Computing (2023) 21:62
https://doi.org/10.1007/s10723-023-09700-y

RESEARCH

A Real-Time Partition Generation Mechanism for Data Skew Mitigation in Spark Computing Environment

Li Yang · Xiong Xiao · Xuedong Zhang · Zhechang Hu · Zhuo Tang

Received: 10 October 2022 / Accepted: 10 October 2023 / Published online: 31 October 2023

Abstract Due to the limitation of the computing power of a single node, big data is usually processed on a distributed parallel processing framework. The data in the real scene is usually not evenly distributed. Data skew will seriously affect the performance of distributed parallel computing, causing excessive load on some tasks and idle computing resources. To solve the above problems, we propose an optimization method based on step size sampling, which can more accurately predict the distribution of intermediate data. Then, we propose a balanced partitioning strategy based on adaptively adjusting computational granularity (BPAG). The adjustment of the computation granularity focuses on the characteristics of sampled data and the usage of computing resources. The balanced partition strategy distinguishes keys with different weights through weighted round-robin and efficient hashing. A partitioning strategy based on high-weight keys (HWKP) and a partitioning strategy based on low-weight keys (LWKP) are proposed. Finally, we implemented BPAG on Spark 2.4.8. We conduct comparative experiments based on four widely used big data benchmarks and five related works in the experimental evaluation. The evaluation results show that BPAG can effectively achieve partition balance and reduce task execution time.

Keywords Data sampling · Data skew · Distributed computing · Partition · Granularity adjustment

L. Yang · X. Xiao (✉) · Z. Hu · Z. Tang
Hunan University, Changsha 410082, China
e-mail: xx@hnu.edu.cn

L. Yang
e-mail: yanglixt@hnu.edu.cn

Z. Hu
e-mail: hzccancer@hnu.edu.cn

Z. Tang
e-mail: ztang@hnu.edu.cn

X. Xiao · Z. Hu · Z. Tang
National Supercomputing Center in Changsha, Changsha 410082, China

X. Zhang
Beijing Institute of Space Mechanics & Electricity, Beijing 100094, China
e-mail: z_xuedong@163.com

1 Introduction

The advent of the information age has produced a data explosion, and the rapid increase in the amount of data has challenged the processing speed of computers. Single-node computing can no longer meet the growing demand for massive application data [1,2], and various distributed parallel computing frameworks such as MapReduce [3] have emerged. MapReduce divides the job into one or more *Map* and *Reduce* stages and stores the data in HDFS [4]. Hadoop [5], Spark [6], and Flink [7] are all concrete implementations based on the concept of MapReduce. Hadoop mainly performs batch processing tasks, while Spark and Flink support batch processing and streaming processing tasks [8]. Spark is an open-source distributed big data processing framework widely used after Hadoop [9]. Take Spark as an example, it includes Cluster Manager and Worker nodes. The Cluster Manager executes on the Spark cluster, controls and manages the Worker nodes of the cluster, and ensures the smooth operation of the system. The Worker runs on the cluster, and as a computing node is responsible for receiving tasks from the Cluster Manager node and sending status reports to the Cluster Manager node. The specific details related to the architecture of Spark are shown in Fig. 1 [10].

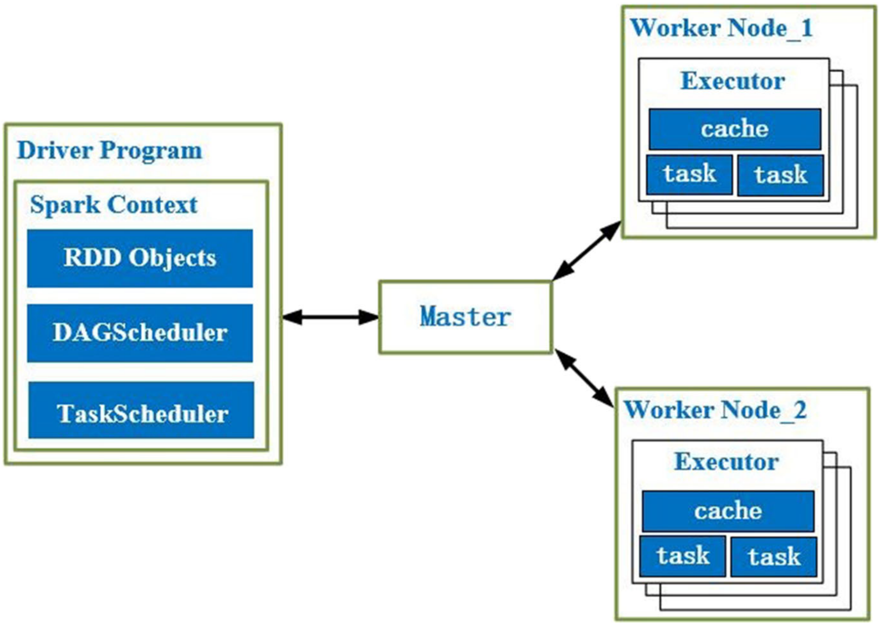
In the real scene data set, the distribution of keys is not evenly distributed. As a result, the data size of each partition in the shuffle process varies greatly, leading to different workloads for each task. If a task has a relatively long execution time, we call it a straggler. The presence of stragglers will increase the overall processing time of the application. In addition, it will require more computing resources. When the system submits a new task, this situation will also lead to idleness and load imbalance of other computing resources [11].

In the distributed parallel computing framework, a shuffle process will be executed between the stages to complete the redistribution of data. The process of data repartitioning and data transmission in the shuffle has a high overhead, which has become the performance bottleneck of the distributed computing framework. In previous research, the optimization of the shuffle stage was mainly carried out from four perspectives, namely data locality, communication cost, resource utilization, and balanced partition [12]. However, the dimensions of the research are relatively single. Although this will significantly simplify the complexity of the system and algorithm design, it cannot fully use the overall computing power of the distributed system.

This article comprehensively considers the impact of resource utilization and balanced partitioning on shuffle in a distributed computing framework. As shown in Fig. 2, the computing resources in the cluster include 10 cores. For a stage with a parallelism of 15, the cluster resources are not enough to complete all tasks at once, and the tasks in the stage need to be executed in two batches. The green boxes represent the tasks executed in the first batch, and the yellow boxes represent the tasks executed in the second batch. At t_1, all cores are busy. At t_2, only some cores are busy, and the rest are idle. Moreover, when the tasks of the second batch have been executed, some of the tasks of the first batch have not yet been completed. The completion time of the stages is much longer than the average completion time of each core, which eventually increases the total time for completing tasks in all stages. As shown in Fig. 2, *Finish Line* is much larger than *Average*.

Through the above observations, we have concluded. First, improperly set computing granularity will lead to low utilization of computing resources. Second, the skew of the intermediate data will lead to inconsis-

Fig. 1 Spark architecture



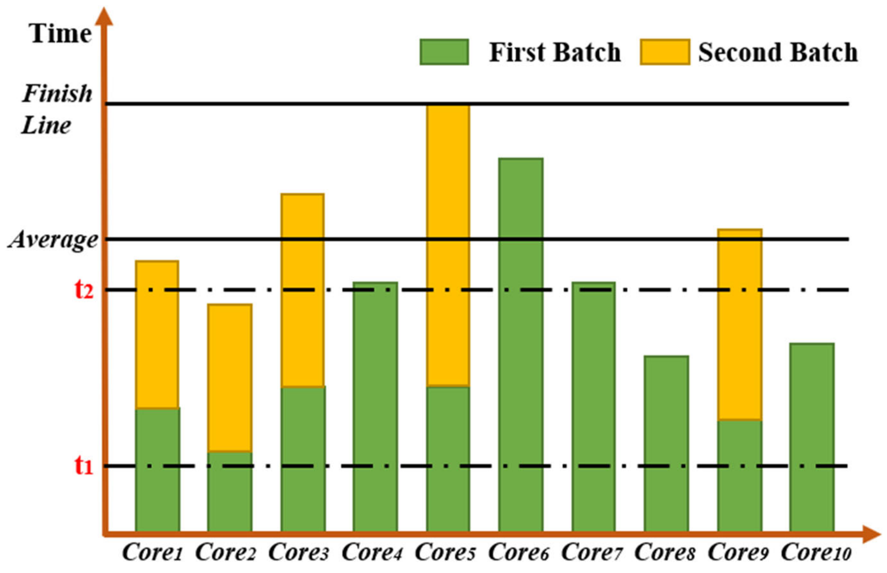


Fig. 2 Data skew and computational resource imbalance

tencies in the time to complete the task on each core. Therefore, an excellent partitioning strategy needs to consider the problem of the unbalanced distribution of computing resources and how to alleviate data skew. This strategy requires an appropriate set of calculation granularity and an optimized partitioning algorithm. The main contributions of this paper are summarized as follows:

- We propose a balanced partitioning strategy based on adaptive adjustment of computational granularity (BPAG). This strategy considers two aspects of resource utilization and balanced partitions to optimize the load balancing of the partitions.
- We propose an optimized sampling method based on step size. It can accurately predict the distribution of the intermediate data output by the map and lay a statistical foundation for the repartitioning method of differential processing. In addition, it can make the hash table used for the auxiliary partition more lightweight to avoid memory overflow.
- We propose a method to adjust the calculation granularity dynamically. By balancing the partition size of the current task, the CPU always has tasks to execute instead of being idle, which improves the resource utilization of the current stage.
- We propose a balanced partitioning strategy, including a partitioning strategy based on high-weight keys (HWKP) and a partitioning strategy based on low-weight keys (LWKP). It uses the weighted round-robin algorithm and the efficient hash algorithm for repartitioning, which can avoid data skew problems and optimize the operating efficiency of applications.

The rest of the paper is organized as follows. Section 2 introduces the work related to data skewness and partition granularity in the distributed parallel computing framework. Section 3 presents the overall system framework of BPAG. Section 4 introduces the step-based sample algorithm and the adjustment model of partitioned calculation granularity. Section 5 proposes a balanced partition strategy, including HWKP and LWKP. Section 6 provides experimental results and evaluation. Section 7 summarizes the article.

2 Related Work

In distributed parallel computing frameworks based on MapReduce, the shuffle is the bridge between the map stage and reduce stage. The shuffle process is accompanied by a large amount of disk I/O and network transmission, so the shuffle significantly impacts the entire application's performance. In previous research [12], the optimization of the shuffle process in the MapReduce framework is carried out from four aspects: improving data locality, optimizing the communication process of shuffle, improving resource utilization, and balancing partitions. These researches are focused on optimizing the shuffle process in one dimension. The data generated in reality is irregular. Data skew will seriously affect the performance of the distributed computing framework, such as extending the entire execution time and idle resources. Many researchers optimize the task of data skew in a distributed environment.

For the data skew algorithm of the shuffle process, Guo et al. [13] proposed an algorithm, iShuffle. While the intermediate data is generated on the map side, it is pushed to the corresponding reduce side. This method can flexibly schedule tasks with load balancing in mind, but the scope of the application is limited. Tang et al. [12] proposed SKRSP, an algorithm for key and partition redistribution to solve skewness. This method can effectively balance the distribution of data, but does not consider the problem of computing granularity, which may still cause tasks to be executed in batches, thereby affecting program efficiency.

Aiming at the operation in the distributed database, Xu et al. [14] proposed a method for dealing with the inner-join, called PRPD. This method combines hash-based and duplication-based strategies to effectively solve the data skew problem of the inner-join. However, applying inner-join directly to outer-join is very complicated and expensive. Cheng et al. [15] proposed a new algorithm called QC to solve the skewness of outer-join in a distributed architecture. The algorithm is based on APGAS. Nevertheless, it can only deal with the data skew in the join operation, which is not universal.

In addition, some research uses distributed parallel scheduling to achieve load balancing. Zheng et al. [16] proposed a fine-grained data parallel processing framework for horizontally expanding the parallelism of task execution. However, this method only optimizes resource utilization for the process of fetching data from HDFS. Zeng et al. [17] focused on the parallel implementation of the K-means on Flink for high-dimensional sparse vector samples. However, this method is limited to specific algorithms and requires the use of streaming calculations. Liu et al. [18] proposed a new partitioner to balance the intermediate data between buckets in Spark. Based on Spark streaming, the data characteristics of each batch can be obtained, and the key distribution of the next batch can be predicted using the sample data that just arrived. He et al. [19] proposed DS^2 to deal with data skew within and between nodes based on data stealing. The above methods can only handle the data skew of streaming data and unbalanced allocation of computing resources, but the support for batch processing is not ideal.

3 System Overview

In the distributed computing framework (e.g., Spark and Flink), tasks executed in parallel are not always executed in an ideal state. If the granularity of the partition is appropriately adjusted, the current stage can be executed in batches. At the same time, through a suitable partition strategy to balance the data size of each partition, the system will better utilize computing resources in batches and avoid long-term tasks. After the previous batch of tasks is completed, the next batch of tasks can be supplemented immediately. In this way, it is possible to keep computing resources as busy as possible while reducing the degree of data skew, thereby improving the unbalanced distribution of computing resources and optimizing the execution time of the entire stage.

In this article, we propose a balanced partitioning strategy based on adaptive adjustment of computing granularity. The overall execution framework of our system is shown in Fig. 3. The main steps are as follows.

Data Sampling. In order to obtain the data distribution of the stage, we need to sample the data in the partition. We propose an optimized step-based sampling algorithm to make it more stable in the case of data skew. The algorithm can sample in different partitions in parallel and obtain the general distribution of the intermediate data according to the sampling results.

Partition Granularity Adjustment. By modeling according to the characteristics of the intermediate data and the computing resources in the cluster, the recommended computing granularity of the current stage can be obtained. The calculation granularity can divide the tasks of the current stage into several complete batches. The computing resources of each batch are busy.

Balanced Partition Strategy. We divide the intermediate data into high-weight and low-weight keys according to the estimated frequency of key occurrence. Based on the consideration of the balance between keys, we propose HWKP and LWKP.

Partition Implementation. The shuffle mechanism in the distributed computing framework will generate an orderly data file and index file for the map task. The key-value pair in intermediate data will obtain a new

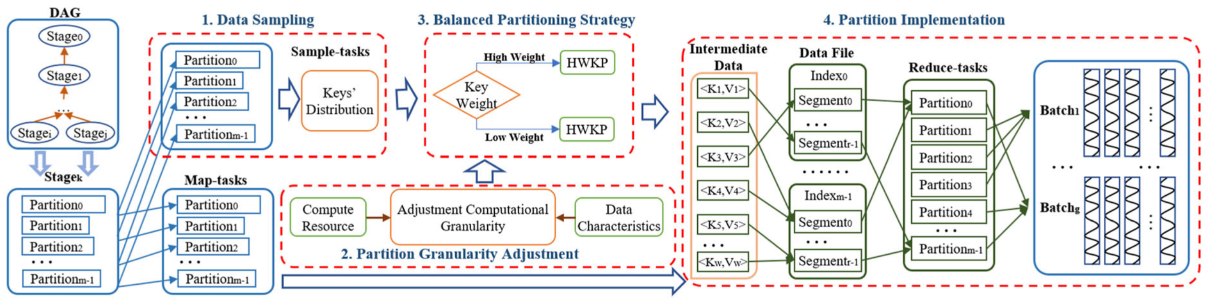


Fig. 3 The framework of BPAG

split id according to the *getPartition* method and dispatch to the corresponding reduce task. Based on this mechanism, LWKP and HWKP are applied in the process of repartition in the shuffle. Finally, the partitions can be evenly distributed in different batches, reducing the overall completion time of the program.

4 Data Sampling and Partition Granularity Adjustment

This section mainly explains the data skew model in Section 4.1 and declares some related variables. We introduce the optimized sampling algorithm based on step size in Section 4.2 to consider different repartitioning strategies to reduce the overhead of the shuffle process. We describe the partition granularity adjustment mechanism in Section 4.3 to alleviate the problem of unbalanced computing resource allocation.

4.1 Data Skew Model

In this model, to quantify the intermediate data obtained by the reducer, we formally declare some variables related to the data skew and some intermediate results and present them in Table 1.

C_K is a positive integer, indicating the number of tuples whose key is K. n represents the number of distinct keys in the intermediate data, q represents one of the keys, and $0 \leq q < n$. $A_{i,j}$ is a two-dimensional matrix used to represent the number of tuples from M_i and received by R_j. M_i will generate tuples of different keys as intermediate data, and R_j will also receive several different key-value tuples according to the default mechanism.

In general, the number of keys of the original input data follows the Zipf distribution [20]. The parameter σ indicates the degree of skew, and the value range is 0.1 to 1.2. It is a constant for a given data set. The more severe the skewness, the greater the value of σ. Without loss of generality, we use $A^{\sigma}_{i,j}$ to denote the number of tuples that R_j gets from M_i when the skewness is σ. All data transferred between the mapper and reducer is the total number of tuples of all keys. The relationship between C_K and $A_{i,j}$ can be formulated as follows:

$$\sum_{q=0}^{n-1} C_{K_q} = \sum_{i=0}^{m-1}\sum_{j=0}^{r-1} A_{i,j} \tag{1}$$

SC_j is the number of tuples received by R_j. In order to make the model more general, we use the skewness σ to describe the number of tuples received by R_j as SC^{σ}_j, which can be calculated as (2):

$$SC^{\sigma}_j = \sum_{i=0}^{m-1} A^{\sigma}_{i,j} \tag{2}$$

Based on the above description and discussion, we can calculate the average number of tuples received by all reducers, an important indicator for calculating the degree of data dispersion. As shown in (3):

$$\overline{mean_{\sigma}} = \frac{\sum_{q=0}^{n-1} C_{K_q}}{p} = \frac{\sum_{j=0}^{r-1} SC^{\sigma}_j}{p} = \frac{\sum_{i=0}^{m-1}\sum_{j=0}^{r-1} A^{\sigma}_{i,j}}{p} \tag{3}$$

Where p is the number of reducers.

Table 1 Variable declaration

$M_i, 0 \le i < m$	M_i:the i^{th} mapper;
	m: the number of mappers;
$R_j, 0 \le j < r$	R_j:the j^{th} reducer;
	r: the number of reducers;
n	the distinct keys;
$K_q, 0 \le q < n$	the q^{th} key;
$\sigma, 0.1 \le \sigma \le 1.2$	an indicator to measure the skewness;
C_K	the number of tuples whose key is K;
$A_{i,j}$	the number of tuples from the M_i and received by the R_j;
SC_j	the number of tuples received by R_j;
$\overline{mean_\sigma}$	the average number of tuples received by all reducers;
std	the standard deviation for reducers;
Fos	an indicator to measure the load balance of reducers.

The standard deviation reflects the degree of dispersion of the data set and the load balance between the reducers. The standard deviation std_σ can be defined as follows:

$$std_\sigma = \sqrt{\frac{\sum_{j=0}^{r-1}(SC_j^\sigma - \overline{mean_\sigma})^2}{p}} = \sqrt{\frac{\sum_{j=0}^{r-1}(\sum_{i=0}^{m-1} A_{i,j}^\sigma - \frac{\sum_{i=0}^{m-1}\sum_{j=0}^{r-1} A_{i,j}^\sigma}{p})^2}{p}} \tag{4}$$

Naturally, if $|SC_j^\sigma - \overline{mean_\sigma} > std|$ is satisfied for R_j, which indicates that the intermediate data will be skewed, and the load of each reducer will be unbalanced. The difference between SC_j^σ and $\overline{mean_\sigma}$ can be calculated by (5):

$$(SC_j^\sigma - \overline{mean_\sigma}) = \sum_{i=0}^{m-1} A_{i,j}^\sigma - \frac{\sum_{i=0}^{m-1}\sum_{j=0}^{r-1} A_{i,j}^\sigma}{p} \tag{5}$$

In order to measure the load balance between reducers, it is necessary to quantify the skewness of the partition. We define *Fos* (Factor of Skew) [21], which is a standard coefficient of variation to characterize the degree of skew between partitioned data:

$$Fos = \frac{std_\sigma}{\overline{mean_\sigma}} \tag{6}$$

It can be seen that the smaller the value of *Fos*, the better the load balance of the reducer, and the smaller the skew of the partition data.

4.2 Optimized Sampling Algorithm Based on Step Size

In order to ensure that the shuffle data balanced partition algorithm can work normally, we need to obtain the key distribution of the intermediate data in advance. Most sampling methods in distributed frameworks only sample intermediate data based on sorting operators to guide the scope division of each key. Since there are distinct keys for intermediate data, if the same repartitioning strategy is handled for keys with different weights, the cost of repartitioning will be huge. Therefore, we consider using different repartitioning strategies for keys with different weights. Its purpose is to apply the expensive repartitioning strategy to fewer keys to reduce the cost of the shuffle process.

This paper proposes a sampling algorithm based on importance. Importance describes the preference of sampling, and the algorithm is biased towards sampling the keys with high frequency. We define $Random(a, b)$ as the important parameter. When deciding whether to accept the step size, the important parameter will be added to the key that frequently appears in the step size to increase the probability of the key being sampled. The algorithm dynamically adjusts the probability parameter of the next sampling according to the frequency statistics of the key being sampled so that the sampling result is more sensitive to the frequently appearing keys in the intermediate data. After the sampling is completed, the data obtained in the sampling result is filtered and counted. The counts of keys obtained by the sampling method are defined as high-weight keys, and the counts of the remaining unsampled keys are defined as low-weight keys. The sampling process is executed in parallel, and this feature can optimize the efficiency of the sampling process. Sampling will take up part of the program running time, but the results after sampling are significant for guiding the adjustment of partition granularity and data repartitioning.

The idea of sampling is the rejection sampling algorithm based on step size proposed in [22]. We adapted the decision mechanism of whether to accept the step size in the algorithm idea. The algorithm performs sequential sampling in an average of $O(n)$ time and only generates n uniform random variables. Some critical parameters involved in the sampling process are shown in Table 2. These parameters are for a specific partition and can be extended to all partitions.

Figure 4 describes the overall process of an optimized sampling algorithm based on step size, which is divided into the following steps:

1. Randomly generate a step size s_i;
2. Based on the hash table T_{hash}, determine whether the value pointed to by the step size satisfies the acceptance condition based on T_{hash}, and if yes, perform Step (3), otherwise return to Step (1);
3. Skip s_i key-value tuples and sample the $(s_i+1)^{th}$ key-value tuple;
4. Add the $(s_i+1)^{th}$ key-value tuple to the sampled data.
5. Maintain T_{hash}, then update kv_i and r_i. If both kv_i and r_i are greater than 0, perform Step (1) to continue the next sampling. If kv_i is equal to 0 or r_i is equal to 0, then there is no key-value tuple available for sampling or the required key-value tuples have been sampled, and the sampling process ends.

Table 2 Sampling parameters

i	a specific partition;
s_i	the step size of random sampling in the i^{th} partition;
len_i	the number of key-value pairs in the i^{th} partition;
kv_i	the number of key-value pairs unprocessed in the i^{th} partition;
num_i	the number of key-value pairs sampled in the i^{th} partition;
r_i	the number of key-value pairs to be sampled in the i^{th} partition;
k_j,	the number of distinct keys;
$0 \le j < q$	one of keys;
v_j	the count of k_j;

For the optimized sampling algorithm proposed in this paper, the essential part is the generation of the step size and determining whether to receive the step size. In this paper, we choose the step generation method described in [22], which generates a random number, and the sampling step size s_i is generated as shown in (7):

$$s_i = \lfloor kv_i(1-u^{1/r_i})\rfloor \tag{7}$$

Where r_i represents the total number of key-value pairs that need to be sampled in the i^{th} partition. kv_i represents the total number of key-value pairs that have not been sampled in the i^{th} partition. The variable u is a random number between 0 and 1. In the sampling process, r_i sample pairs need to be extracted from kv_i not sampled key-value pairs.

During the sampling process, we maintain a hash table T_{hash} in the context of all partitions, which is used to store the keys and their frequencies appearing in the sample data. There are two situations to determine whether to accept the step: if $\sigma_1+\sigma_2 \ge 0.5$, then the step is accepted, otherwise the step is rejected. σ_1 is a random number between 0 and μ_1. The definition of σ_2 is as follows:

$$\sigma_2 = \begin{cases} \dfrac{v_j}{\sum_{j=0}^{q-1} v_j}, & v_j - \dfrac{\sum_{j=0}^{q-1} v_j}{q} \le 0 \\[2ex] \dfrac{v_j}{\sum_{j=0}^{q-1} v_j} + Random(0,\mu_2), & v_j - \dfrac{\sum_{j=0}^{q-1} v_j}{q} > 0 \end{cases} \tag{8}$$

Where $Random(a,b)$ is a function that takes a random number on $[a,b)$. v_j is the value of the element k_j referenced in the current step in T_{hash}, which represents

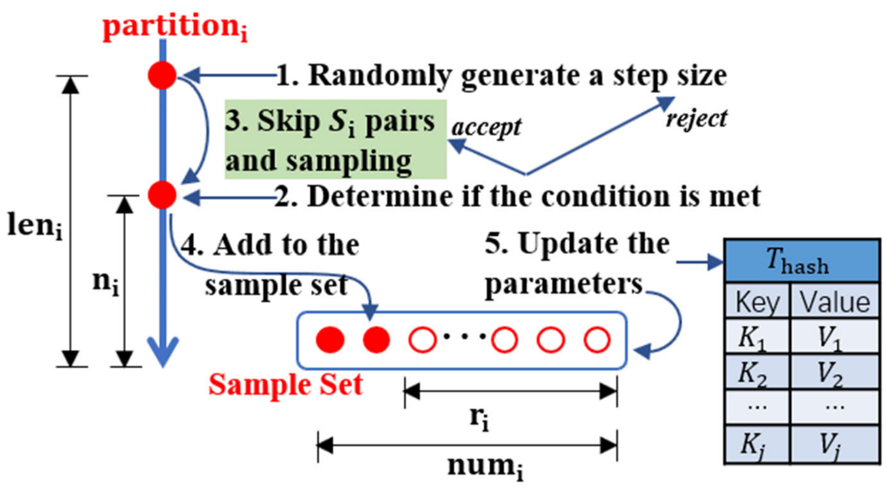


Fig. 4 The process of importance-based sampling

the number of occurrences of k_j in the partition during the sampling process. When v_j is less than the average of all keys, it means that k_j has appeared less frequently so far, so we choose to filter out this key. Otherwise, this means that k_j has appeared more frequently so far, so we add a random number between 0 and μ_2 as an additional weight to make it easier to obtain the key through sampling. The relationship between the variables μ_1 and μ_2 is: $\mu_1 \leq \mu_2$ and $\mu_1 + \mu_2 \geq 1.2$. After the sampling is completed in this way, it can be used for the subsequent balanced partition strategy.

Algorithm 1 describes the process of sampling data on the map side. Each partition is sampled in parallel. The results of parallel sampling of each partition are sent to the driver for aggregation, where T_{hash} is visible to each partition.

Algorithm 1 Optimized Sampling Based on Step Size.

```
Input: Map side partition: $Partitions$.
Output: A hash table with a key-value pair (key, key frequency): $T_{hash}$; The number of samples obtained by real sampling: $Count$.
1: $T_{hash} = \emptyset$;
2: for Each partition$_i$ in $Partitions$ that participates in sampling do
3:     Initialize the position of the sample, $pos = 0$;
4:     Initialize the number of unprocessed key-value pairs $kv_i$;
5:     Initialize the number of key-value pairs to be sampled $r_i$;
6:     Initialize the number of samples sampled $count_i = 0$;
7:     while $n_i > 0$ and $r_i > 0$ do
8:         Randomly generate step size $s_i$;
9:         Randomly generate variable $\sigma_1$;
10:        Generate variable $\sigma_2$;
11:        if $\sigma_1 + \sigma_2 \geq 0.5$ then
12:            Accept the step size;
13:        else
14:            Reject the step size;
15:        end if
16:        if $accept$ then
17:            Initialize a pointer $cur = pos + s_i$;
18:            $T_{hash} = T_{hash}.add(k_{cur}, v_{cur})$;
19:            $pos = pos + s_i + 1$;
20:            $n_i = n_i - s_i - 1$;
21:            $r_i = r_i - 1$;
22:            $count_i = count_i + 1$;
23:        end if
24:    end while
25: end for
26: $Count = \bigcup_{i=0}^{m-1} count_i$;
27: return ($T_{hash}$, $Count$).
```

4.3 Partition Granularity Adjustment

The task is the fundamental unit for performing calculations, and it uses partitions as the granularity of calculations. The granularity of the partition directly determines the running time of the task. On the one hand, too large partition granularity leads to too few partitions, which increases the amount of data that each partition needs to deal with. In addition, too few partitions may cause some nodes to be unable to be assigned tasks and waste computing resources. On the other hand, too small a partition granularity can lead to more partitions, which increases the number of computing tasks. When computing tasks exceed the number of available cores, the number of batches executed by tasks will increase [10,23]. Thread switching consumes a lot of resources and delays the completion time of the entire application. Therefore, we need to dynamically adjust the partition granularity in an appropriate way to solve the problem of unbalanced computing resource allocation in the distributed computing framework.

The computing granularity represents the size of the partition processed by each calculating task, and can also be referred to as the partition granularity. During the running of the application, partitions are divided according to the number of data blocks in the storage or fixed parameters. In this way, the amount of data in each partition is not entirely equal. This granularity division method is seriously lacking in adaptability. The executor will execute each task. If the workload of a task is much greater than the average, the completion time of the entire job will be delayed. Unreasonable computing granularity settings cause the appearance of stragglers. If the proportion of stragglers is too large, the granularity setting is unreasonable and needs to be adjusted. The calculation granularity here is for a particular stage in the job. During the sampling process, we detect and count the tasks of the current stage on all executors. Some parameters obtained by process statistics are shown in Table 3.

According to Table 3, the average number of key-value pairs that need to be executed by each core A_p can be calculated by (9).

$$A_p = \frac{p_c}{Clu_c} \tag{9}$$

In the case of batch execution, A_p is expressed as the sum of the number of key-value pairs that each core in multiple batches should execute. The number of key-value pairs is only for sampled data, not for all intermediate data. It reflects the ideal computing load that each CPU core should carry when the partition data is entirely unbalanced.

$V_{\max}$ is the frequency of the most common key in the sampled data. If $V_{\max} > A_p$, it means that the key-value pair contained in the key cannot be executed in one batch and must be executed in batches. In order to enable the key-value pairs contained in the key to be executed in perfect batches, then $V_{\max}$ should be a multiple of the number of key-value pairs processed in each batch of each computing task. In addition, in the case of partition balance, it must be satisfied that A_p is a multiple of the number of key-value pairs processed in each batch of each computing task. In summary, the greatest common divisor of $V_{\max}$ and A_p is defined as B_p, which represents the optimal number of key-value pairs processed by each computing task. The calculation method of B_p is expressed by (10).

$$B_p = GCD(V_{max}, A_p) \tag{10}$$

It is worth noting that when $V_{\max}$ and A_p are both prime numbers and not the same, the greatest common divisor of these two numbers is 1. This value will cause the number of partitions to be too large and lose the meaning of granularity adjustment. Therefore, in this case, the algorithm will deal with the $V_{\max}$ and A_p, adjust their values to approximate composite numbers and make the greatest common divisor of the two numbers not less than the threshold φ. In most cases, the threshold φ is equal to multiple quotients of p_c and Clu_c. Therefore, the optimal partition number N_p of the current stage when the application is run can be calculated by (11).

$$N_p = \frac{p_c}{B_p} = \frac{p_c}{GCD(V_{\max}, A_p)} = \frac{p_c}{GCD(V_{\max}, \frac{p_c}{Clu_c})} \tag{11}$$

Table 3 Variable declaration

N_p	advised number of partitions;
Clu_c	the number of executors in the cluster;
p_c	the number of key-value pairs in the sampled data;
t_c	the number of distinct keys in the sampled data;
V_{max}	the frequency of the most common key in the sampled data;
A_p	the number of key-value pairs that each core should handle;

5 Balanced Partitioning Strategy

This section proposes a balanced partitioning strategy and reallocates resources according to the computational granularity proposed in Section 4.3. We define low-weight keys and high-weight keys, considering the balance of partition settings between keys with different weights. The weighted round-robin algorithm in load balancing and the efficient hash algorithm are applied to the repartition method.

The common partition algorithms are *HashPartitioner* and *RangePartitioner*. *HashPartitioner* applies to most non-sort operations; *RangePartitioner* will sort the key, and then divide the key into *numPartitions* collections according to the range, which applies to most sort operations.

The hash-based partitioner can uniformly repartition the data of different keys but does not consider the skew between keys. It will cause too much data in a specific partition, thereby delaying the completion time of the program. In order to deal with the situation where the keys are skewed, we need to classify the weight of keys in T_{hash} and use different algorithms to redistribute. We organize the data distribution estimated by sampling, and according to the Pareto principle, we refer to keys in the intermediate data that appear in the first $n\%$ of frequencies as high-weight keys, and keys that appear in the last $(100 - n)\%$ of frequencies as low-weight keys. Where n is a positive integer from 0 to 100. We keep high-weight keys in T_{hash} and filter out low-weight keys. The distinction between high-weight key and low-weight key in specific implementation is whether they exist in T_{hash}.

For high-weight keys, we use a weighted cycle-based method for partitioning. Round-robin-based packet schedulers usually have low complexity and provide long-term fairness [24]. The weighted round-robin is an algorithm for polling weighted packets. Since the algorithm only works on high-weight keys, the types of keys in the memory are significantly reduced, and the memory pressure caused by round-robin operations with high time complexity can be alleviated. For low-weight keys, we use the method based on Murmurhash [25] to partition, which can optimize the time efficiency of the key hashing process. This method fills the remaining partitions after the high-weight key is assigned so that the amount of data in each partition is closer. Finally, we apply the partition strategy to the *getPartition* method to balance high-weight keys and low-weight keys. By guiding the repartitioning in the shuffle process, the partition is globally balanced, and the running time of the entire program is shortened.

Two kinds of balanced partition algorithms based on low-weight keys and high-weight keys are proposed in this paper. The algorithms perform their respective duties and work together in the data skew scene to achieve the balanced partition.

5.1 Partition Strategy Based on High-weight Keys

This section proposes a partitioning strategy based on high-weight keys (HWKP). In order to distinguish it from the partition on the map side, this section uses split to refer to the partition on the reduce side. This partition is only a logical partition before the policy is generated and does not contain any data. To describe the HWKP in detail, we define the following data structures:

(1) Np. According to the recommended number of partitions obtained in the granularity adjustment module, we need to divide the intermediate data into Np splits to guide the repartitioning process.

(2) T_{hash}. A hash table with a (key, value) structure is used to store the key distribution of the sampled data. The frequency of each key is the number of times the key appears. It can be expressed as (12):

$$T_{hash} = \bigcup_{k \in C} (k, k.count) \tag{12}$$

Where k is a key in the set of sampled data, and $k.count$ is the number of times the key appears during the sampling process. C is the key set in the sampled data.

(3) T_{fre}. A hash table with a (key, value) structure is used to store the key distribution estimation of intermediate data. After several sampling values are obtained by sampling and stored in T_{hash}, we can estimate the approximate distribution of the intermediate data according to T_{hash} and the overall sampling rate of the sampling process. $fre(i,k)$ is the number of occurrences of k in the i^{th} partition. fre_k is the number of occurrences of k in all partitions, as described in (13):

$$fre_k = \sum_{i=0}^{m-1} fre(i,k) \tag{13}$$

Where m is the number of partitions on the map side. The relationship between T_{fre} and fre_k can be expressed by an (14).

$$T_{fre} = \bigcup_{k \in C} (fre_k / ratio) \tag{14}$$

We can obtain the approximate distribution of high-weight keys through T_{fre}, which is convenient for load balancing operations on partitioned data. $Ratio$ is the probability of sampling.

(4) W. A data set is used to record the weight of each split in the intermediate data. W_j records the weight of the j^{th} segmentation of the intermediate data. This flag indicates whether there is any space left in the split. The default value of this flag is *false*. The initial value of W can be calculated as Eq. (15):

$$W_j = \frac{\sum_{k=0}^{C.size-1} (fre_k/ratio)}{N_p} \tag{15}$$

W_j is calculated by dividing the total number of key-value tuples of intermediate data by the number of splits. It is related to the number of key-value pairs allocated to each split in the partition table. $Ratio$ is the probability of sampling. If t key-value pairs are assigned to a split, the weight of the split will decrease by t. W_{avg} records the average weight of all splits, which can be calculated as (16).

$$W_{avg} = \sum_{j=0}^{N_p-1} W_j/N_p \tag{16}$$

(5) R. A hash table is used to store the repartition numbers of high-weight keys. The algorithm will poll W for each key in T_{fre}, and each poll will get the $split_m$ with the largest weight in W to match the k_j in T_{fre}, and then add the new key-value tuple $(k_j, split_m)$ into R.

As shown in Fig. 5, the process of HWKP can be divided into the following three steps:

1. Obtaining high-weight keys and estimating the distribution of high-weight keys based on the sampled data;
2. Calculating the weight of each split in the intermediate data, weighting each split and adapting it to the key of the intermediate data to determine the partition number of each high-weight key, and updating the weight of split;
3. Implement *HWKP* in the shuffle stage.

First, the sampling process will filter low-weight keys that appear less frequently, obtain high-weight keys, and count the number of occurrences fre_i. We predict the distribution of intermediate data based on the sampling rate and store it in T_{fre}. W is a set that records the weight of each split in the intermediate data. We poll W to get the most weighted split in W and the tag of *false*. We match the split with the key in T_{fre} and add the key-value tuple (key, split) to R to form a partitioning strategy. R is used to guide repartitioning during the shuffle stage, and at the same time, update the weight of the corresponding split with W. Assuming that the key-value tuple currently stored in R is (A, $partition_i$), the weight in W can be updated according to (17).

$$W_i = W_i - fre_A/ratio \tag{17}$$

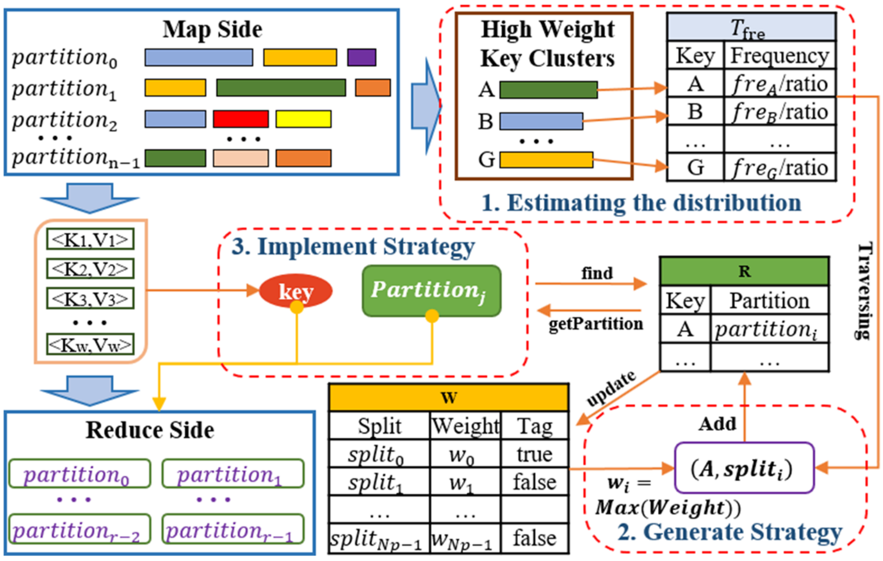


Fig. 5 The steps of high-weight keys balanced partition algorithm

Where $ratio$ is the probability of sampling. The weight of the split represents the ability of the current split to accommodate the key-value tuples. The higher the weight, the more key-value pairs that can be accommodated. Therefore, we prioritize the partitioning of keys into high-weight splits, which is essentially a greedy-based strategy. When the updated W_i satisfies $W_i \geq W_{avg} * tol$, where $0.9 \leq tol < 0.95$, it means that if we continue to divide the key-value tuple into i^{th} split will cause data skew, which is something we should try to avoid. When W_i exceeds the threshold, we set the flag of the i^{th} split to *false*, indicating that the repartition number is no longer pointed to the split. In the reduce side of the shuffle stage, a query operation is performed in R for each key. If the key is included in R, it will be repartitioned with the strategy in R. If not, it will be repartitioned by the partition method based on low-weight keys.

5.2 Partition Strategy Based on Low-weight Keys

In this section, a partitioning strategy based on a low-weight key (LWKP) is proposed. In order to better describe the LWKP, we define the following specific data structure in the model:

(1) D. A pseudo-random number generator generates a series of pseudo-random numbers. During collision processing, a random number of the sequence is taken as the starting point, and the number D_i in the sequence is selected as the arithmetic number of addition and modulus.

(2) W_j. W_j records the weight of the j^{th} split of the intermediate data, and the initial value is the split weight updated after repartitioning the high-weight keys.

(3) W_b. The balance value of all splits, which can be calculated by (18):

$$W_b = W_{avg} \times tol \tag{18}$$

where tol is tolerance, $0.95 \leq tol < 1$. This value is used to determine whether a split is balanced. The purpose of adding the coefficient tol is to reserve space for the current key assigned to the split so that all key-value pairs of the current key assigned to the split can still ensure that each split is balanced.

(4) T_c. A hash table for handling hash collisions. The table stores a conflicting key and its new partition number obtained according to the pseudo-random detection method.

As shown in Fig. 6, the implementation of LWKP can be divided into the following three steps:

1. Obtain the balance value of all partitions;
2. Traverse the conflict table. If the key is included in the table, the partition number will be returned directly. If the key is not included in the table, *Murmurhash* calculation will be performed on each key of the intermediate data to get a new partition number. If the key exceeds the balance value, it will be considered as a conflict. The pseudo-random detection method will process the conflict to get a new partition number and add it to the conflict table.
3. Implement *LWKP* in the shuffle stage.

In the process of outputting the intermediate data to the reduce side, we will get the new partition number through *LWKP*. First, we obtain the approximate distribution of the key for the intermediate data through an optimized step-based sampling method. The sampling results obtained in Section 4.2 are used to distinguish high-weight keys and low-weight keys. Keys not included in T_{fre} are lightweight keys. Assume that the low-weight key currently to be processed is K.

Second, check whether there is K in T_c. If it exists, it means that the key has conflicted before and the conflict has been resolved, so the $partition_i$ in T_c is directly returned as the new partition number. If there is no K in T_c, it means that the key did not conflict before. *Murmurhash* can be used to calculate the new partition number, and the calculation result is t. If the weight of t^{th} split satisfies $W_t + 1 \leq W_b$, it means that t^{th} split still has room for key-value tuples, and t is regarded as the new partition number; If the weight of the new partition number satisfies $W_t + 1 > W_b$, the t^{th} split does not have enough space to accommodate the key-value tuple with the key K. That is, for collisions, we can use pseudo-random detection methods to handle collisions. The new number of partitions is calculated by (19):

$$p = (t + D_i)\%N_p \tag{19}$$

where D_i is an element in the pseudo-random number sequence D, and N_p is the advised partition number

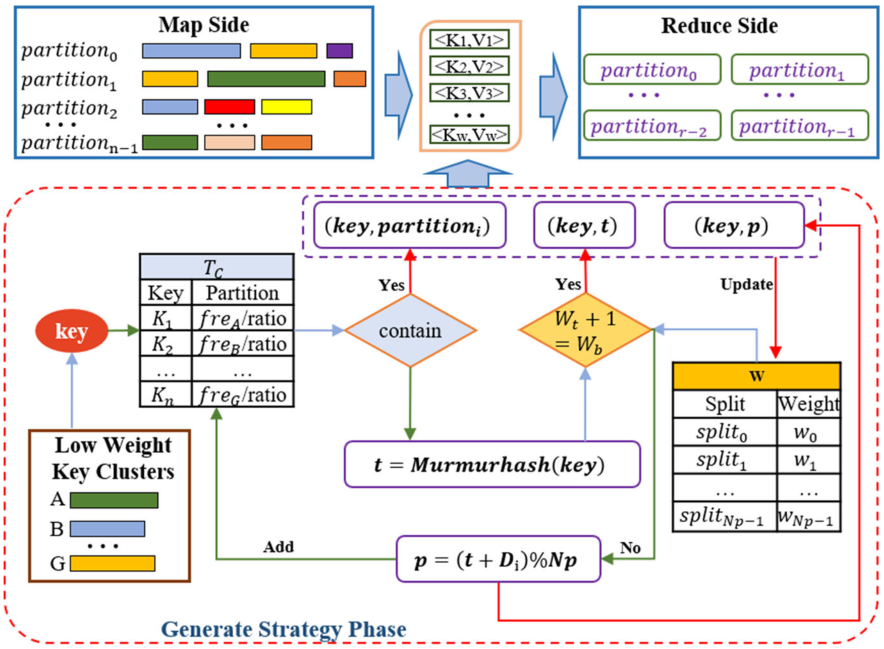


Fig. 6 The steps of low-weight keys balanced partition algorithm

Algorithm 2 Estimation of the Keys' Distribution of the Intermediate Data.

```
Input: A hash table of sampling result for intermediate data: $T_{hash}$.
Output: A collection of key distribution estimate for intermediate data: $T_{fre}$.
1: Initialize the frequency of occurrence of $k$ in the $partition_i$: $fre_{i,k}$.
2: for Each $partition_i$ in map side do
3:     $fre_k = fre_k + fre_{i,k}$;
4: end for
5: for Each $fre_k$ of the frequency of each key in $T_{hash}$ do
6:     //Calculate the estimated value of the current key;
7:     $T_{fre_k} = fre_k/ratio$;
8: end for
9: return $T_{fre}$.
```

obtained through the granularity adjustment module and the size of W.

Finally, we obtained $(K, partition_i)$, (K, t) or (K, p) as the final strategy to guide the repartitioning through the above steps and update the weight of the corresponding split in W.

5.3 Balanced Partition Algorithm Based on High Weight Keys

For the HWKP, we can first get the approximate distribution of intermediate data through the optimized step-based sampling algorithm, and at the same time, we can also get the occurrence frequency of high-weight keys in the intermediate data.

Based on the above data, Algorithm 2 traverses each partition of the map side and calculate the frequency of

Algorithm 3 Obtaining the Repartition Number of the High-weight Key.

```
Input: A table of the key distribution estimation of the intermediate data: $T_{fre}$; A collection of split weight of the intermediate data: $W$.
Output: The collection of high-weight keys repartition number: $R$.
1: Initialize the weight of each split: $W_j$;
2: Initialize the average weight of each split: $W_{avg}$;
3: for Each key in $T_{fre}$ do
4:     Sort $W$ in descending order which the tag of element is false;
5:     //Get the most weighted index of split in $W$;
6:     $index_{max} = i$;
7:     //Add a key-value pair to the collection $R$;
8:     $R.add(key, i)$;
9:     //Update the weight of the $split_i$ in $W$;
10:      $W_i = W_i - fre_A/ratio$;
11:      //If the weight exceeds the threshold, it will no longer be polled;
12:      if $W_i \geq W_{avg} * tol$ then
13:          $tag_i = true$;
14:      end if
15: end for
16: return $R$.
```

occurrence of k in the i^{th} partition, then aggregate the data on the driver and estimate the distribution of the intermediate data according to the sampling rate.

Through Algorithm 2, we can obtain the distribution estimation set T_{fre} of intermediate data. Then, Algorithm 3 is designed to generate the repartition strategy through processing the data of each key in T_{fre} and obtain the repartition number of high-weight keys. W is the split weight of intermediate data, which contains the weight and tag of each split. The weight represents the ability of the split to hold the key-value pair, and the

Algorithm 4 Obtaining the Repartition Number of the Low-weight Key.

```
Input: A low-weight key: $K$; An updated collection of split weight of the
    intermediate data: $W$; A sequence of pseudo-random: $D$.
Output: The index of repartition: $P_r$.
1: Initialize the table for conflict handling: $T_c$;
2: Initialize the weighted round-robin temporary variable: $p$;
3: Initialize the balance value of all splits: $W_b = W_{avg} * tol$;
4: if $T_c.contains(K)$ then
5:     //Update the weight of $W$;
6:     $W_{T_c.get(K)} = W_{T_c.get(K)} + 1$;
7:     $P_r = T_c.get(K)$;
8: else
9:     //The result of K's Murmurhash;
10:    $t = Murmurhash(K)$;
11:    if $W_t + 1 \leq W_b$ then
12:        $W_t = W_t + 1$;
13:        $P_r = t$;
14:    else
15:        //$W_t$ exceeds the balance value for conflict handling;
16:        $p = (t + D_i)\%N_p$;
17:        $T_c.add(K, p)$;
18:        $W_p = W_p + 1$;
19:        $P_r = p$;
20:    end if
21: end if
22: return $P_r$.
```

tag indicates whether the split can still accommodate key-value pairs. First, after initializing W and W_{avg}, and then traverse each key in T_{fre}, we can select the $split_i$ with the highest weight and the tag of *false* in W, add (key, i) to R and update the weight of $split_i$ in W at the same time. If $W_i \geq W_{avg} * tol$ is satisfied, the tag of $split_i$ can be set to *true*. Finally, we get the repartitioned set R of the high-weight key and the updated weight set W. R is used to guide the repartition of high-weight keys, and W is used as the input of the repartition strategy of low-weight keys.

5.4 Balanced Partition Algorithm Based on Low Weight Keys

For the LWKP, the condition judgment is the most critical part. We need to jump to the appropriate strategy for repartitioning according to the condition branch. Algorithm 4 describes the process in detail.

Firstly, we initialize the pseudo-random sequence D, obtain the updated split weight set W of the intermediate data, and initialize the balance value W_b of all splits. Whenever we get a low-weight key, we first check whether the key is included in T_c. If it is included, we could directly return the repartition number and update the weight of the split in W. If it is not included, the *Murmurhash* method will be used to calculate the result t. If $W_t + 1 \leq W_b$ is satisfied, it will return the repartition number and update the weight in W. Otherwise, the pseudo-random detection method is adopted, and the repartition number is obtained by adding a pseudo-random sequence and performing a modulo operation. Finally, the repartition number is added to T_c, and the weight in W is updated.

5.5 Allocation Implementation

The shuffle mechanism in the distributed computing framework generates an ordered data file and index file for the map tasks. The key-value pairs in the intermediate data obtain a new split id according to the *getPartition* method, and then dispatch to the corresponding reduce task for execution.

Based on this mechanism, we apply LWKP and HWKP to the repartitioning process in the shuffle and finally make the partitions evenly dispersed in different computing batches, reducing the overall completion time of the program. Algorithm 5 shows the process of applying the generated partition strategy to shuffle. First, we check whether the key is included in the high-weight partition table. If it is, it means that the key is a high-weight key, and we can get the repartition number by querying the repartition table for the high-weight key. If it does not, it means that the key is a low-weight key, and we need to make a conditional branch judgment, jump to the corresponding branch, and get the final repartition number.

6 Experiments

We first introduce the related experimental setting in Section 6.1. Second, we verify the effectiveness of our proposed method in Section 6.2 by overall performance evaluation. Finally, we compare and analyze the performance results by using multiple benchmarks and multiple related strategies in Section 6.3.

6.1 Experiment Setting

This section has designed several experiments to verify the effectiveness of sampling methods and partitioning strategies. In order to reuse the existing task scheduling and memory management mechanisms in the distributed computing framework, we choose Spark as the distributed experimental environment. Based on

Algorithm 5 The Implement of BPAG.

```
Input: A key of intermediate data: $K$; A table for conflict handling: $T_c$; A table for repartitioning high-weight keys: $R$.
Output: The index of repartition: $P_r$.
1: Initialize the result of $Murmurhash$: $t$;
2: Initialize the weighted round-robin temporary variable: $p$;
3: if $R.contains(K)$ then
4:     $P_r = R.get(K)$;
5: else
6:     if $T_c.contains(K)$ then
7:         $P_r = T_c.get(K)$;
8:     else
9:         //The result of K's Murmurhash;
10:        $t = Murmurhash(K)$;
11:        if $W_t + 1 \leq W_b$ then
12:            $P_r = t$;
13:        else
14:            //$W_t$ exceeds the balance value for conflict handling;
15:            $p = (t + D_i)\%N_p$;
16:            $P_r = p$;
17:        end if
18:    end if
19: end if
20: return $P_r$.
```

Spark 2.4.8, the evaluation is carried out in a real cluster of AliCloud, which contains eight workers and one master. See Table 4 for specific hardware and software configuration. The evaluation data in this experiment is stored on HDFS. By modifying the core modules in the Spark source code, the sampling method and partitioning strategy are applied to the original Spark framework. All experiments are based on the *standalone* mode, using Spark's default parameter configuration to complete.

In order to quantify the impact of the BPAG proposed in this paper, as shown in the Table 5, four practical application scenarios that are prone to data skew are selected. This paper uses the benchmarks in Hibench [26] (WordCount, Join, PageRank, and KMeans) to experimentally evaluate the impact of BPAG on the degree of data skew and the overall running time of the program. We get the average processing time by repeating each program three times.

Table 5 Benchmark types

Application types	Benchmarks
Simple applications	WordCount
Distributed database operations	Join
Search engine applications	PageRank
Social network applications	KMeans

In our experiment, three partitioning algorithms are selected for comparison experiments, namely Hash (*HashPartitioner* [27]), SKRSP [12], and SP (SPPartitioner [18]). In addition, to evaluate the performance of the execution time of PBAG, we have added a comparative experiment with EC (*EC-shuffle* [28]) and DS^2 [19].

6.2 Overall Performance Evaluation

This section independently conducts experimental evaluations on the sampling method, calculation granularity adjustment, and partition strategy in BPAG. The necessity and complementarity of the three steps are verified through experiments.

6.2.1 Sampling Method Evaluation

BPAG predicts the distribution of intermediate data through sampling. The accuracy of sampling directly affects the generation of partition policies, which in turn affects the overall execution time of applications. In the experiment, the sampling method in this paper is compared with the sampling method in *RangePartitioner* and FAST [22]. Its accuracy is measured by the

Table 4 Configuration of the cluster

Node Type	Hardware	Software
Master (1)	Intel Xeon CPU E5-2670, 4 Cores@2.60 GHz, 32 GB, 1000 Mb/s Ethernet	Hadoop 2.7.0, JDK 1.8, Ubuntu 12, Spark 2.4.8
Worker (8)	Intel Xeon CPU E5-2670, 4 Cores@2.60 GHz, 8 GB, 1000 Mb/s Ethernet	Hadoop 2.7.0, JDK 1.8, Ubuntu 12, Spark 2.4.8

MSE (Mean squared error) of all keys. The calculation method is as follows:

$$MSE = \sqrt{\frac{\sum_{k \epsilon F} (E_k - R_k)^2}{|F|}} \tag{20}$$

where F is the collection of all keys, $|F|$ is the number of distinct keys, E_k is the estimated frequency of occurrence of a particular key, and R_k is the actual frequency of occurrence of a particular key. When a key is not sampled, the value of R_k is zero.

We consider that in the real application scenario, the data distribution of each map partition is not necessarily uniform. In order to accurately measure the universality of the sampling algorithm, we divide the partition data into two types: uniform partition and skewed partition. The occurrence frequency of strings in uniform partitions is a uniform distribution, and the occurrence frequency of strings in a skewed partition is the Zipf distribution ($\sigma = 0.8$). In order to accurately reflect the efficiency of the system, the data size of this experiment is set to 2GB, a moderate volume. We first count the actual word frequency of the input data, then call the sampling algorithm to estimate the number of occurrences of each key, and use *MSE* as an indicator to measure the accuracy of the sampling process. This indicator can show the difference between the actual word frequency and the estimated word frequency. The smaller the value of *MSE*, the smaller the difference between the actual word frequency and the estimated word frequency, which means the higher the accuracy of the sampling.

The experiment generates two kinds of data sets, the uniform data set and the skewed data set. Each type of data set generates two data sets containing 5000 keys and 50000 keys. For more clarity, we set different sampling rates for comparison experiments. In addition, if the sampling rate is set too high, the amount of data that needs to be left in the memory during the sampling process will be too large. Memory overflow is not conducive to the normal conduct of the experiment and the effective observation of the effect. Therefore, only 3% of the sampling rate is selected as the standard in this experiment, and a comparison experiment with the sampling rate of 15% is organized. The experimental results are shown in Table 6. When the sampling rate increases, the number of samples that the sampling algorithm can collect increases, so the original data's estimation is more accurate.

For the same input data, when there are more distinct keys in the data set, the number of data contained in each key will be reduced accordingly, and the distribution of keys will be more even, which is beneficial to the estimation of sampling results. It can be observed from the experimental results that the proposed algorithm is significantly better than the other two sampling methods. When the data is even, our method can perform undifferentiated sampling for all keys. When the data is skewed, the importance-based sampling algorithm can predict the intermediate data more accurately. Therefore, our method is more adaptable and can effectively sample data under different conditions.

6.2.2 Partition Granularity Adjustment Testing

To test the necessity of adjusting the calculation granularity, we choose *partitionBy* as the benchmark and conduct comparative experiments before and after the particle size adjustment. Because *partitionBy* only involves repartitioning operations, there is no addi-

Table 6 Accuracy comparison

Method	Rate	Balanced 5000	B-50000	Skew 5000	S-50000
BPAG	3%	1075	194	436	143
Range	3%	1452	324	637	154
Random	3%	1543	269	587	182
FAST	3%	1493	299	648	176
BPAG	15%	304	163	226	67
Range	15%	608	219	342	94
Random	15%	329	158	327	87
FAST	15%	523	177	334	90

tional overhead, and it is suitable for observing the overall impact of the granularity adjustment on the program when the slope of the input data changes. The overall impact of the granularity adjustment on the partitioning strategy can be observed by checking the deviation between the map partitions and the average execution time of the program.

In the experiment, we use a 2GB data set to observe the fluctuation of *Fos* and the average processing time when the slope of the input data increases. Figure 7(a) describes the experimental results of *Fos*, and the results show that the dynamic adjustment of granularity can relieve the data skew between map partitions to a certain extent. However, regardless of whether the granularity is adjusted, the application will always use the partition algorithm to balance the partitions. Only changing the partition granularity cannot significantly optimize the skewness between the mapping partitions.

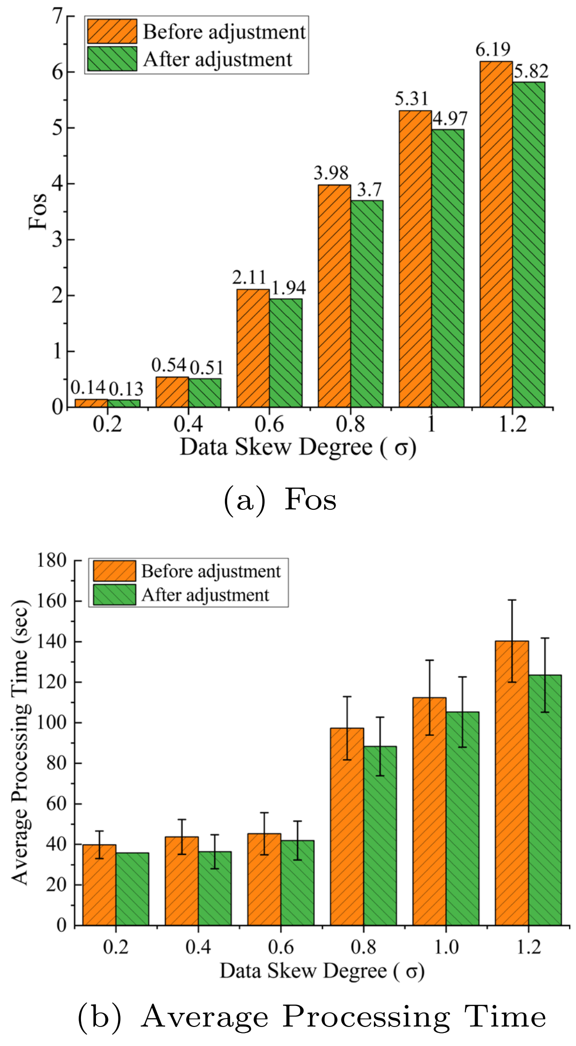


Fig. 7 The effect of partition granularity adjustment

Figure 7(b) is the experimental result of the average processing time of the program. In order to eliminate the interference of the partitioning algorithm on the granularity adjustment algorithm, the experiment uses the system default partitioner to partition without balancing the partition. On the whole, as the skewness of the input data increases, the program's execution time increases accordingly. When $\sigma < 0.6$, due to the adjustment of the calculation granularity, the calculation tasks can be completed in batches, thereby avoiding the occurrence of long tasks, and the average execution time of the program has been smoothly shortened. Obviously, as the data skewness increases, the partitioning algorithm is not used to partition the data evenly, and long tasks will appear at this time. At this time, the execution time of the application begins to increase significantly.

In addition, we added the observation of the degree of granularity after using the partitioning algorithm to adjust the granularity of the partition. The experimental results show that although the number of partitions after granularity adjustment is different under different shew degrees, they are all within an interval, that is, 3 to 4 times the total number of cores of the nodes in the cluster [29].

6.2.3 Partition Strategy Testing

In order to observe the impact of the partitioning strategy on data skew, we organize statistical experiments on the data distribution. The experiment takes *partitionBy* as an example to count the data size of each partition after repartitioning. We will compare BPAG with *HashPartitioner* for experimental analysis. Since the number of partitions may be affected by the granularity adjustment, we set the number of partitions to 36, the total number of cores in the cluster. The input data obeys the Zipf distribution with $\sigma = 1.0$.

The experimental results are shown in Fig. 8. When using *HashPartitioner* to partition, because the input data is skewed, the repartition number is only obtained by simple hash calculation, so the partition data is still skewed. The data ratio of each partition obtained by *HashPartitioner* is not uniform. The ratio of the partition with the most data to the partition with the least data

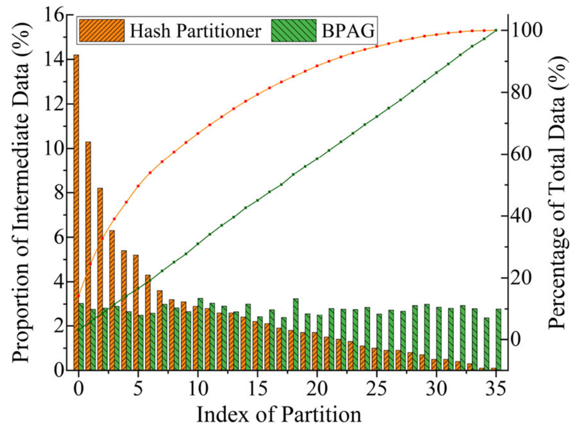


Fig. 8 The proportion of intermediate data in all partitions

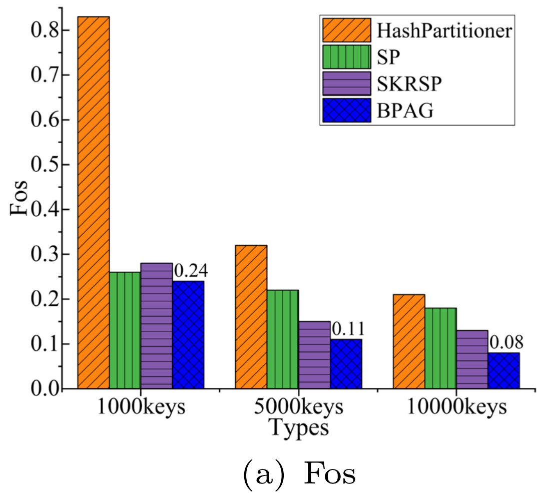
is greater than 14%. After using the BPAG method for repartitioning, the data distribution between partitions is significantly more even, so our partitioning method can alleviate the data skew problem more effectively.

6.3 Benchmarks' Experimental Analysis

In order to evaluate the overall effectiveness of BPAG, this section chooses the three algorithms (Hash, SKRSP, and SP) and two related works (EC and DS^2) proposed in Section 6.1 as comparative experiments. According to the characteristics of the experiment, the experimental results of the BPAG algorithm are analyzed in the four benchmark tests of WordCount, Join, PageRank and KMeans.

6.3.1 WordCount Performance Test

WordCount is a classic application in Spark, which mainly involves the operation of *reduceByKey*. This experiment creates a custom partitioner for the *reduceByKey* operation and uses the BPAG strategy to adjust the calculation granularity and repartition. The *reduceByKey* operation aggregates keys, so the experimental results are only related to the type and number of keys. Three data sets of 1,000 keys, 5,000 keys, and 1,000 keys with a size of 5GB are used in the experiment, and the data obeys the Zipf distribution ($\sigma = 0.8$). Figure 9 shows the statistics of Fos and program execution time of *HashPartitioner*, SKRSP, SP and BPAG in the *WordCount* benchmark.



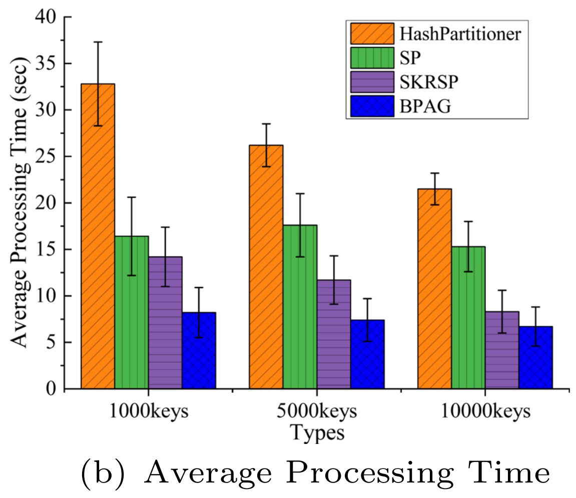


Fig. 9 WordCount benchmark with different keys

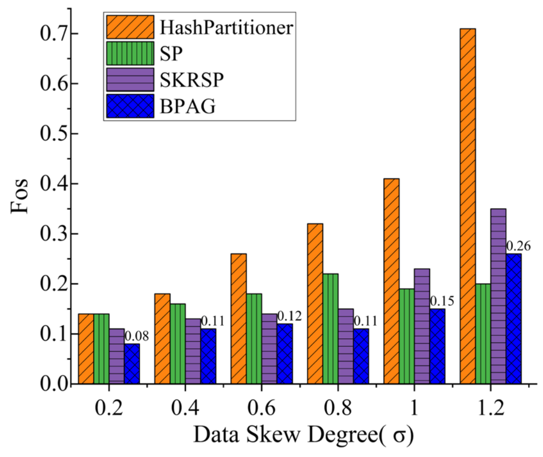
It can be seen from Fig. 9(a) that the more distinct keys, the better the skew between partitions. Because when the total amount of data remains the same, the increase in the number of types makes the difference in the number of key-value pairs contained in each key more minor. Compared with the other three control groups, the Fos of BPAG is relatively low, indicating that our method has a better effect. Figure 9(b) shows that as the number of distinct keys increases, the average execution time of each algorithm decreases to varying degrees. The execution time of BPAG is always the shortest, followed by SKRSP. SP is relatively stable for repartition optimization of different inclinations. The more serious the data skew, the more pronounced the effect. Nevertheless, the execution time of BPAG is not sensitive to the type of key. Because the partitioning algorithm makes the partitioned data more balanced, the execution time of each computing task has a slight difference, and the overall running time of the program is significantly shortened.

Figure 10 shows the performance evaluation of *HashPartitioner*, SKRSP, SP and BPAG under different data skewness. The test data is Zipf distribution data with skewness varying between 0.2 and 1.2, and its size is 5GB. It can be seen from Fig. 10(a) that as the data skewness increases, the *Fos* of *HashPartitioner* is increasing. The other methods *Fos* are always in a relatively stable state. Under the same skewness, BPAG's *Fos* is always better than SKRSP. When the skewness is small, the data itself is relatively uniform, and even after the repartitioning process, it will not produce obvious effects. When σ exceeds 0.8, the Fos of SP is lower than SKRSP, but the overall execution time of its task exceeds SKRSP. Generally speaking, as the data skewness increases, the cost of the repartitioning strategy of *HashPartitioner* is relatively small, but there is still serious data skewness, which leads to longer execution time. Other algorithms have made great improvements in skew optimization, but they are not stable enough. BPAG has significantly improved the repartition optimization and the overall execution time of the task. Under the same degree of skew, BPAG has about 20% to 50% performance improvement over SKRSP.

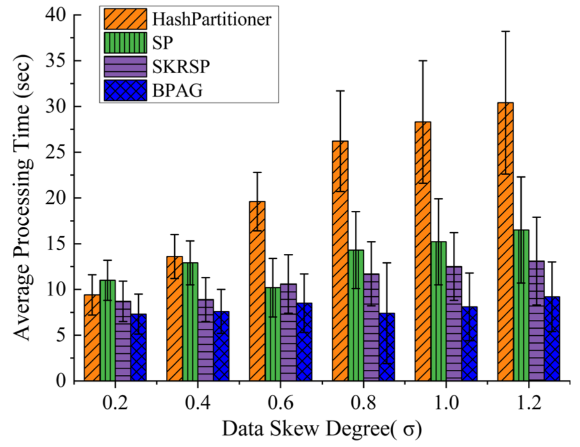
Figure 11 shows the performance comparison of relevant algorithms on data sets of size 1TB, 2TB, and 3 TB respectively, when the degree of skew is 0.8. As can be seen, BPAG still maintains performance advantage compared with HashPartitioner, SKRSP, and SP. In particular, when the data size is 3TB, BPAG can reduce the average processing time by 40%, 27% and 21% compared with HashPartitioner, SKRSP, and SP, respectively.

6.3.2 Join Performance Test

In this section, we use the *Shuffle Join* operation to evaluate the performance of BPAG. In order to run the *join* application, the data set we used includes a large table that conforms to the Zipf distribution (σ from 0.2 to 1.2) and a small table that conforms to a uniform distribution. We set up two test cases: a) A large table joins another large table ($400M \times 400M$); b) A large



(a) Fos



(b) Average Processing Time

Fig. 10 WordCount benchmark with different data skew

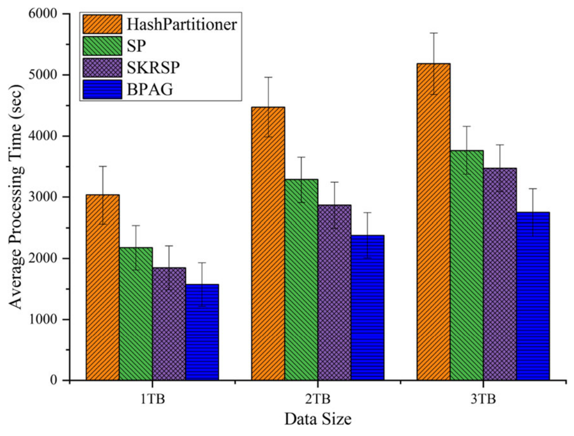
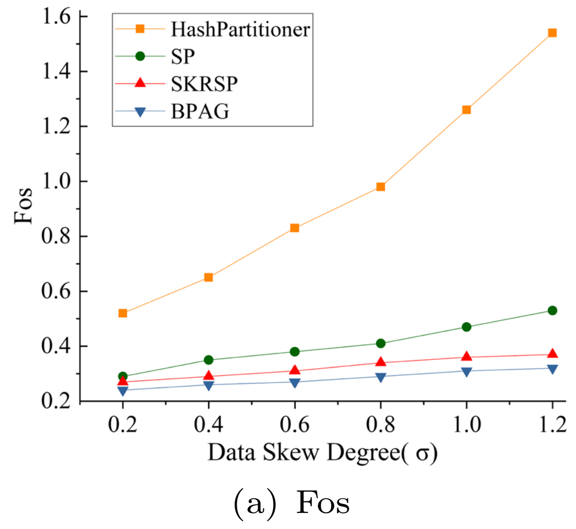


Fig. 11 WordCount benchmark with different data size

table joins a small table ($2G \times 20M$). Figures 12 and 13 show when the σ table varies from 0.2 to 1.2 under two different *join*.

From the comparison of Figs. 12(a) and 13(a), each algorithm has a better load balancing effect on the first type of test case. Because the original data of the second test case is more skewed, this also has a direct manifestation in the execution time of the task. From the comparison between Figs. 12(b) and 13(b), with the increase of σ, the processing time under *HashPartitioner* rises sharply. *Join* (2G×20M) is very sensitive to changes in data skewness, and the growth curve is relatively steep. The BPAG, SKRSP, and SP curves change relatively smoothly and always maintain a low numerical level.



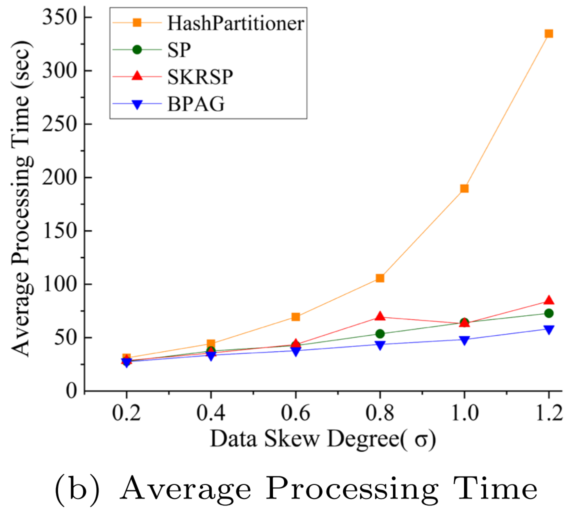
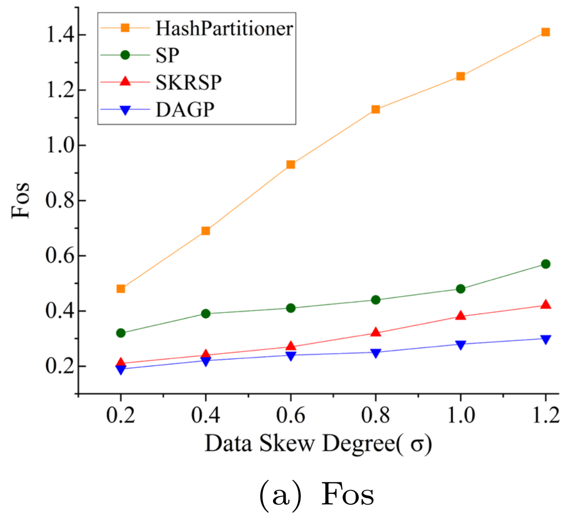


Fig. 12 Join benchmark with different σ (400M×400M)



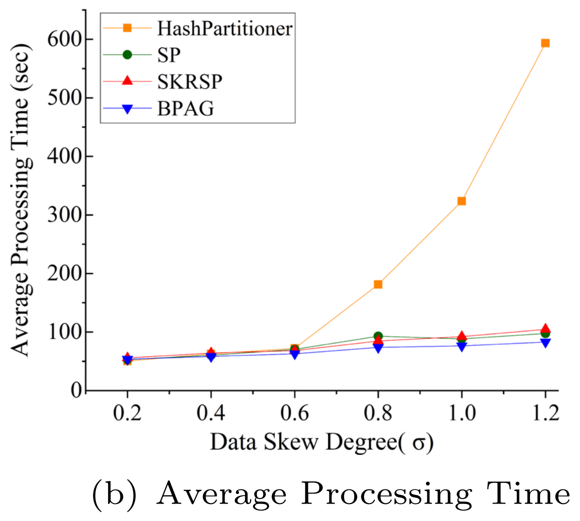


Fig. 13 Join benchmark with different σ (2G×20M)

During the running of the application, there are some long tasks in *HashPartitioner* that are much higher than the average, which delays the running time of the entire program. The other three strategies (SKRSP, SP, and BPAG) have completed partition data balancing, so there will be almost no obvious long tasks. However, the calculation granularity of the partition may not match the amount of data that a single CPU can process. Tasks in SKRSP and SP need to be executed in batches, which delays the execution time of the entire application. It can be directly seen from the statistical results that the optimization effect of BPAG is more significant in the case of severe data skew (the second test case). Therefore, the BPAG proposed in this paper effectively handles data skew and adjusts the calculation granularity of partitions, effectively avoiding program delays caused by task batch execution, so our method is more stable and efficient.

6.3.3 PageRank Performance Test

PageRank is the algorithm Google uses to rank the relevance of search results. It records the importance of each page by counting the number and weight of page links. The benchmark uses three data sets from SNAP: web-Stanford (2,312,497 edges), web-Google (5,105,039 edges) and web-BerkStan (7,600,595 edges) [30]. In this experiment, we have carried out experimental statistics on the five control tasks in Section 6.1, and the experimental results are shown in Fig. 14. Compared with *HashPartitioner*, although EC and DS^2 do not equalize the skew of partition data in the shuffle process, they can reduce the execution time of computing tasks through communication optimization and task scheduling. Other partition optimization strategies can reduce the execution time of the program to a certain extent. The BPAG proposed in this paper balances the partition data while considering adjusting the calculation granularity, making it significantly reduce the program execution time.

6.3.4 KMeans Performance Test

KMeans is a well-known clustering algorithm used for knowledge discovery and data mining. The benchmark test uses three data sets from the Stanford Network Analysis Project [31]: Higgs-twitter, LiveJournal, and Facebook Social Network (Facebook). In this experiment, we have performed experimental statistics on the five control tasks in Section 6.1, and the experimental results are shown in Fig. 15. Compared with *PageRank*, the optimization effect of DS^2 on *KMeans* is not as good as other strategies. The iterative calculation makes its tasks generate much overhead in the rescheduling process. BPAG still maintains a perfect optimization effect, and its performance is more prominent in Facebook data. The partition data balancing operation in shuffle also significantly improves the acceleration of iterative calculations.

7 Conclusion

In the distributed computing framework, data skew will seriously affect the performance of distributed applications, such as prolonging the entire execution time and idle resources. In this article, we implement an optimization method based on step size sampling to solve this problem, which can more accurately predict the distribution of intermediate data. We propose

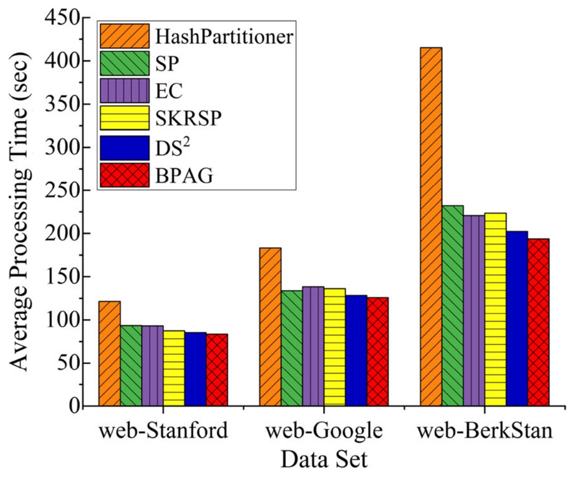


Fig. 14 PageRank benchmark with different data sets

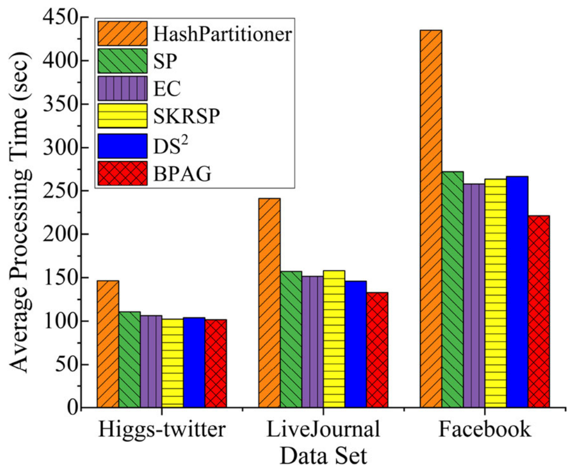


Fig. 15 KMeans benchmark with different data sets

a balanced partitioning strategy based on adaptively adjusting computational granularity (BPAG) to alleviate the unbalanced distribution of computing resources and data skew issues. Experimental results show that the method proposed in this paper has better evaluation results under multiple performance metrics. In addition, the partitioning algorithm improves program performance more significantly in scenarios where the input data volume is large and the skewness is high. Our work is suitable for batch processing distributed computing framework. In future work, we plan to carry out research work on the stream processing framework.

Author contributions All authors contributed to the design and implementation of the research, to the analysis of the results and to the writing of the manuscript.

Funding The work is supported by the National Key Research and Development Program of China (2021ZD40303), the National Natural Science Foundation of China (Grant Nos. 62225205, 92055213, 62302157), Natural Science Foundation of Hunan Province of China (2021JJ10023), Shenzhen Basic Research Project (Natural Science Foundation) (JCYJ20210324140002006), the Hunan Provincial Natural Science Foundation of China (Grant 2021JJ40612), Natural Science Foundation of Changsha of China (kq2208042).

Availability of data and materials All data generated or analysed during this study are included in this published article.

Declarations

Conflicts of interest The authors declare that they have no competing interests.

References

1. Song, Y., Yang, L., Wang, Y., Xiao, X., You, S., Tang, Z.: Parallel incremental association rule mining framework for public opinion analysis. Inf. Sci. **19**(3), 523–545 (2023)
2. Xiao, X., Li, C., Jiang, B., Cai, Q., Li, k., Tang, Z.: Adaptive search strategy based chemical reaction optimization scheme for task scheduling in discrete multiphysical coupling applications. Appl. Soft Comput. 121 (2022)
3. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
4. hdfs (2021) https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs
5. Hadoop (2014) http://hadoop.apache.org
6. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: Usenix conference on hot topics in cloud computing (2010)
7. Flink (2017) https://flink.apache.org
8. Anusha, K., Usha Rani, K.: Performance evaluation of spark sql for batch processing. In: Emerging research in data engineering systems and computer communications, pp. 145–153 (2020)
9. Cheng, G., Ying, S., Wang, B., Li, Y.: Efficient performance prediction for apache spark. J. Parallel Distrib. Comput. **149**, 40–51 (2021)
10. Apache spark. https://spark.apache.org/docs/3.5.0/cluster-overview.html (2016)
11. Beame, P., Koutris, P., Dan, S.: Skew in parallel query processing. In: 33rd ACM SIGMODSIGACT-SIGART symposium on principles of database systems, pp. 212–223 (2014)
12. Tang, Z., Lv, W., Li, K., Li, K.: An intermediate data partition algorithm for skew mitigation in spark computing environment. IEEE Trans. Cloud Comput. **9**(2), 461–474 (2018)
13. Guo, Y., Rao, J., Cheng, D., Zhou, X.: ishuffle: Improving hadoop performance with shuffleon-write. IEEE Trans. Parallel Distrib. Syst. **28**(6), 1649–1662 (2017)
14. Yu, X., Kostamaa, P., Xin, Z., Liang, C.: Handling data skew in parallel joins in sharednothing systems. In: ACM SIGMOD international conference on Management of data, pp. 1043–1052 (2008)
15. Cheng, L., Kotoulas, S., Ward, T.E., Theodoropoulos, G.: Efficiently handling skew in outer joins on distributed systems. In: 14th IEEE/ACM international symposium on cluster, cloud and grid computing, pp. 295–304 (2014)
16. Zheng, L., Shen, Y.: Improve parallelism of task execution to optimize utilization of mapreduce cluster resources. In: IEEE 17th International conference on computational science and engineering, pp. 674–681 (2015)
17. Zeng, Z., Li, k., Duan, M., Liu, C., Liao, X.: K-means parallel acceleration for sparse data dimensions on flink. In: 2019 IEEE 21st International conference on high performance computing and communications; IEEE 17th international conference on smart city; IEEE 5th international conference on data science and systems (HPCC/SmartCity/DSS), pp. 2053–2058 (2019)
18. 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. Futur. Gener. Comput. Syst. **86**, 1054–1063 (2018)
19. He, Z., Li, Z., Peng, X., Weng, C.: Ds2 : Handling data skew using data stealings over high-speed networks. In: 2021 IEEE 37th International conference on data engineering (ICDE), pp. 1865–1870 (2021)
20. Lin, J.: The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce (2012)
21. Tang, Z., Ma, W., Li, K., Li, K.: A data skew oriented reduce placement algorithm based on sampling. IEEE Trans. Cloud Comput. **8**(4), 1149–1161 (2016)
22. Vitter, J.S.: Faster methods for random sampling. Communications of the ACM **27**(7), 703–718 (1984)
23. Karau, H., Konwinski, A., Wendell, P., Zaharia, M.: Learning spark: lightning-fast big data analysis, O'Reilly Media, Inc. (2015)
24. Yuan, X., Duan, Z.: Fair round-robin: A low complexity packet schduler with proportional and worst-case fairness. IEEE Trans. Comput. **58**(3), 365–379 (2009)
25. Murmurhash. https://en.wikipedia.org/wiki/MurmurHash (2016)
26. Hibench. https://github.com/Intel-bigdata/HiBench (2021)

27. Hashpartitioner. http://spark.apache.org/docs/latest/api/scala/index.html (2017)
28. Yao, X., Wang, C., Zhang, M.: Ec-shuffle: Dynamic erasure coding optimization for efficient and reliable shuffle in spark. In: 2019 19th IEEE/ACM International symposium on cluster, cloud and grid computing (CCGRID), pp. 41–51 (2019)
29. Ousterhout, K., Panda, A., Rosen, J., Venkataraman, S., Xin, R., Ratnasamy, S., Shenker, S., Stoica, I.:The case for tiny tasks in compute clusters. In: 14th Workshop on hot topics in operating systems (HotOSXIV). (2013)
30. Leskovec, J., Lang, K.J., Dasgupta, A., Mahoney, M.W.: Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. Internet Math. **6**(1), 29–123 (2008)
31. Stanford large network dataset collection (2013)