An Intermediate Data Partition Algorithm for Skew Mitigation in Spark Computing Environment

Zhuo Tang, Wei Lv, Kenli Li, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—In the parallel computing framework of Hadoop/Spark, data skew is a common problem resulting in performance degradation, such as prolonging of the entire execution time and idle resources. What lies behind this issue is the partition imbalance, that causes significant differences in the amount of data processed by each reduce task. This paper proposes a key reassigning and splitting partition algorithm (SKRSP) to solve the partition skew from the source codes of Spark-core_2.11 project, which considers both the partition balance of the intermediate data and the partition balance after shuffle operators. First, we propose a step-based algorithm for sampling the input data to estimate the general key distribution of entire intermediate data. According to the types of the specific applications, we design two algorithms: hash based key reassigning algorithm (KRHP) and rang based key splitting algorithm (KSRP), which can generate appropriate strategy and implement the skew mitigation in shuffle phase. KKSRP generates the weighted bounds to split intermediate data for the type of sort-based applications while KRHP records these reassigned keys and the new reducers these keys belong to for other applications. Finally, we implement SKRSP in Spark 2.2.0 and evaluate its performance through four benchmarks exhibiting significant data skew: *WordCount*, *Sort*, *Join*, and *PageRank*. The experimental results verify that our algorithm not only can achieve a better partition balance but also reduce the execution time of reduce tasks effectively.

Index Terms—Data sampling, data skew, MapReduce, partition, spark

1 Introduction

As a fast and general engine for large-scale data processing, Apache Spark [1] divides data into multiple partitions of RDD [2] and processes them stage by stage. The shuffle phase is the basis for dividing the stages in the DAG, which can maintain the read/write relationship between map tasks in map stages and reduce tasks in reduce stages [3]. Compared with Hadoop [4], Spark is a faster data processing platform based memory computing, and has a more efficient implementation mechanism for large-scale data processing.

As the intermediate data usually consist of many key/value tuples in the shuffle phase, keys of tuples in each map task output datum must be allocated to a certain reduce partition according to specific partition methods. These methods are usually used to calculate which reduce partition a key should belongs to [5]. When the data distribution among all the reduce partitions were unbalanced, it may cause remote fetching failure and extra communication overheads, and longer execution time of overload reduce tasks, which result in prolonging of the entire execution time and idle resources. Because when other tasks with smaller data sizes have been completed, the larger one has not been finished yet and the others must wait for its completion [6].

To optimize performance in the MapReduce framework, the previous solutions mainly include the following aspects: Improving the data locality [7], [8]; Reducing the fetching cost for reduce tasks [9], [10]; Improving resource utilization [11], [12], [13]; Improving the partition balance and mitigating the partition skew [10]. Implementing an efficiently balanced partition algorithm to make every reduce partition a similar size is the most direct method because an imbalanced partition is the root cause of the skew problem [14]. Moreover, it changes the partition strategy directly without moving data again after shuffle phase, so that it has no extra transmission overhead. And it is easily integrated with Spark because we apply the partition strategy based on Spark's original mechanism.

Hash partition [15] and *range* partition [16] are the two default partition algorithms in Spark. The hash method is the simplest and is applied to the vast majority of applications in Spark, except for *sort* operations. *Sort* operations need preserving the total ordering, so the *range* partition method is designed for this demand. *Range* can only be implemented on the application whose key ordering has been defined. Unfortunately, both of these two methods easily cause the skewness of intermediate data.

- *K. Li is with the College of Information Science and Engineering, National Supercomputing Center in Changsha, Hunan University, Changsha, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA. E-mail: lik@newpaltz.edu.*
- *Z. Tang, W. Lv, and K. Li are with the College of Information Science and Engineering, National Supercomputing Center in Changsha, Hunan University, Changsha, Hunan, China 410082. E-mail: {ztang, lkl}@hnu.edu.cn, lik@newpaltz.edu.*

Manuscript received 20 May 2018; revised 31 Aug. 2018; accepted 25 Oct. 2018. Date of publication 31 Oct. 2018; date of current version 4 June 2021. (Corresponding author: Zhuo Tang.)
Recommended for acceptance by J. Chen.
Digital Object Identifier no. 10.1109/TCC.2018.2878838

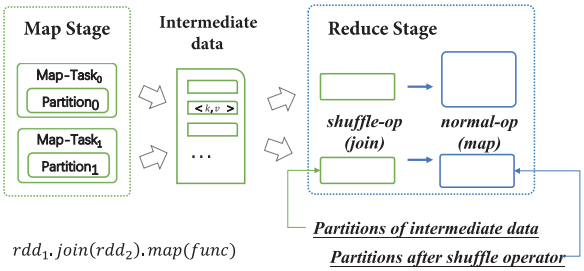


Fig. 1. The integrated balance of reduce task.

To solve the skewness, some useful methods were proposed by previous researches. Most of them were designed for Hadoop and utilized the system characteristics [17], [18], [19], but only a few partition methods is based on Spark [20], [21]. However, being different from Hadoop, Spark can execute the tasks stage by stage. Therefore, some Hadoop based approaches do not necessarily apply to Spark. Most of methods compute partition strategy only on the basis of the tuples number of intermediate data, which is the simplest but thoughtless. To generate more applicable partition policies, previous studies also took the complexity of reducer side algorithms into account, and considered both data locality and fairness.

This work uses *shuffle-op* to represent RDD operators that cause shuffle phase and use *normal-op* to represent the others. Then a reduce task in Spark computing framework contains two parts: *shuffle-op* and *normal-op*. Actually, limited to the traditional Mapreduce framework, previous studies only consider the execution of *shuffle-op*. In the Spark, as show in the Fig. 1, after fetching the splits of intermediate data, a reduce task start processing the data with *shuffle-op* and *normal-op*. However, after *shuffle-op*, the number of tuples in this partition changes and the partitions become skewed. It means that if the shuffle operator makes the tuple numbers change a lot, the balance of intermediate data cannot guarantee the balance of partitions after *shuffle-op*. In this way, a good Spark partition strategy should make a trade off between the balance of *shuffle-op* and that of *normal-op*.

Drawing many inspirations from these ideas, based on a step-based rejection sampling method, this paper proposes the key cluster reassigning and splitting partition algorithms (SKRSP) to relieve the skewness among reducers, which considers the partition balance of both *shuffle-op* and *normal-op*. The main contributions of this paper are summarized below.

- An improved step-based rejection algorithm is proposed and applied to sample the map partitions and estimate intermediate data key distribution according to the switch of map side combination. Based on Spark RDD, this method has a high degree of parallelism and execution efficiency, which can achieve a much better approximation to the key distribution.
- A hash-base key reassigning partition algorithm (KRHP) is proposed to divide intermediate data more fairly to balance the load among the reduce tasks. This partition method is suitable for all the applications except for these sort requiring operations.
- A weighted range partition algorithm based on key splitting (KSRP) is proposed to allocate the intermediate data more evenly and mitigate partition skew among reducers, and it is suitable for applications that require sorting.
- A partitioning algorithm is proposed to split the intermediate data considering both the partition balance of the intermediate data and the partition balance after shuffle operator.

The rest of the paper is organized as follows: Section 2 introduces the survey work on skew mitigation over a parallel computing system based on MapReduce. Section 3 introduces the overall system framework of SKRSP. Section 4 presents the step-based data sampling method and key distribution estimation. Section 5 proposes the partition policy computation, including KRHP and KSRP. The experimental results and evaluations are provided in Section 6. Section 7 concludes the paper.

2 Related Works

For the typically skewed distribution of intermediate data, we must face many real world applications exhibiting significant data skew, including distributed database operations such as *Join*, *Grouping* and *Aggregation* [22], [23], [24], search engine applications (Page Rank, Inverted Index, etc.), and some simple applications (sort, grep, etc.) [4]. Methods by which to handle data-skew effects have been studied previously in parallel database studies [25], [26]. After the release of Hadoop, data skew has also been studied in the MapReduce environment, which is more similar to our work:

Chen et al. [17] presented LIBRA, which is an improved range partition strategy that uses an innovative sampling method to ascertain a highly accurate key distribution of intermediate data by sampling only a small fraction of map tasks. It is an excellent approach for Hadoop. Unfortunately, sampling directly for the intermediate data will bring a huge overhead in Spark actual environment.

Gufler et al. [27] proposed the partition cost model for load balancing in MapReduce, which took into account both skewed data distributions and complex reducer side algorithms. Later, they proposed TopCluster [28], an approximation algorithm for the input data distribution which scales to massive data sets.

Ibrahim et al. [18] developed a novel algorithm named LEEN for locality-aware and fairness-aware key partitioning in MapReduce. They sorted all keys by their fairness/locality value and greedily choose the reduce node with the maximum fairness score for each key.

The above approaches take advantage of Hadoop's features, so they can achieve relatively good results. However, as mentioned in Section 1, they only consider the balance of one shuffle operator. Thus, the advantages of the Hadoop-based algorithms cannot be fully realized in Spark environment. During the past several years, some skew mitigation algorithms specifically designed for spark were proposed.

Some studies focus on a specific application, such as *Join* [29], [30]. Comparing with specialized improvement solutions, Liu et al. presented a more general partition method, SP-Partitioner [21], which makes each bucket which collects the data from the same map task be assigned equal sized data. Unfortunately, it partitions only on the basis of the tuple number of intermediate data.

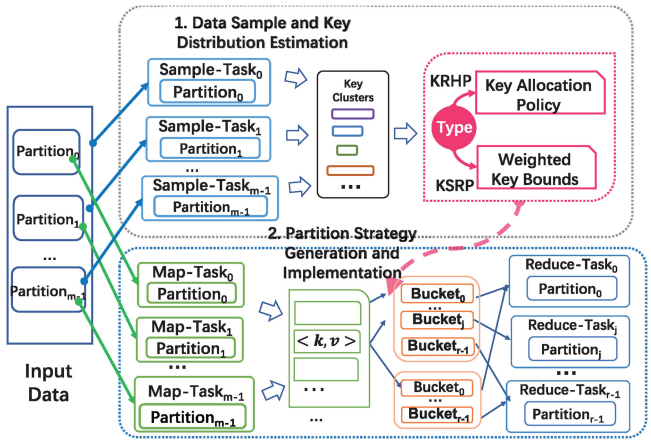


Fig. 2. The Framework of SKRSP.

Yu et al. tackled the data skew problem from a different angle. They proposed SASM [31], which repartitions unprocessed blocks of overload tasks to other idle tasks. This method is similar to the SkewTune [19] based on the Hadoop. Their disadvantages are that during the execution of reducer, the reassignment of unprocessed blocks cost a lot overhead, including extra transmission and computation.

The above work gives us a lot of inspiration, but does not provide an effective solution on Spark. There are two important issues to address: (1) how to estimate the key distribution more accurately and (2) how to calculate the more appropriate partition strategy.

3 System Overview

This section presents the system in a general way. This system implements the key cluster reassigning and splitting approach to solve data skew for general applications. In our implementation, SKRSP is established based on the Spark-2.2.0. In this version, for each map task, each key/value tuple in the map output obtains its reducer index respectively according to its key and the partition method.

In this paper, a more appropriate partition strategy are proposed to improve the partition balance. The architecture of our system is shown in Fig. 2, which contains the following four steps:

Data Sampling. To avoid the data skewness among reduce tasks, it is necessary to estimate the key distribution of intermediate data before the shuffle phase. Therefore, a prior sample job must be launched on the input of map tasks before the regular job. We implement a step-based rejection sampling algorithm on different partitions in parallel.

Key Distribution Estimation. All the samples and corresponding sample rates are collected from different map splits, which constitute the input to calculate the weight of each key by the sample rate. On this basis, we can estimate the general key distribution of intermediate data.

Partition Strategy Computation. In accordance with the specific application scenario of the Spark job, this system adopts different methods to generate the allocation strategy. For these applications that belong to the class of *sort*, the algorithm KSRP is proposed to determine the weighted bounds. And the final key reassignment policies can be obtained by other algorithm KRHP.

TABLE 1
Variable Declaration

len_i	the ith partition size;
num_i	the sampling size for the ith partition;
N_i	the number of unprocessed tuples left in the ith partition;
n_i	the number of tuples that remain to be selected for the sample;
s_i	the random step that counts the number of tuples to skip over.

Intermediate Data Partition. On the basis of the partition strategy obtained from the third step, during the writing phase of shuffle, each key/value $<K, V>$ tuple gets its reduce partition index r and then is written to shuffle files sequentially according to r. For the output of a map task, these $<K, V>$ tuples with the same r compose a bucket, which will become part of the rth reduce partition and be processed by the rth reducer.

The execution of the shuffle phase depends on Spark's original mechanism, and the main contribution of this paper is to provide a way to generate a suitable key allocation strategy according to current input data.

4 Data Sampling And Key Distribution Estimation

4.1 Description

To ascertain the distribution of the intermediate data, a step-based rejection algorithm is first proposed and implemented, which can sample for different partitions with appropriate sampling size in parallel. To adapt to different application scenarios, a variable $ratio$ is used to adjust the proportion of sampled map partitions. For the $ratio$, the number of sampled map tasks is preferably a multiple of worker nodes. Hence if there are m mappers and w workers, $ratio$ is best set to multiples of w/m. Because when the keys are evenly distributed over all the map partitions, the distributions of all intermediate keys in each partition are more or less the same. So in this case, a small ratio is already available for the general key distribution; otherwise, ratio is supposed to be 1.0.

For illustrative purpose, some significant variable declarations are presented in Table 1.

Fig. 3 describes the general process of the step-based rejection sampling algorithm, which can be divided into four steps:

1) Generate a random step s_i;
2) Determine whether to accept the step s_i. If accept, go to Step (3), otherwise return to Step (1);
3) Skip over the next s tuples;
4) Add $(s+1)^{th}$ to the sample. Set $N_i = N_i - s_i - 1$ and $n_i = n_i - 1$. If $N_i > 0$ and $n_i > 0$, return to Step (1), otherwise stop sampling.

First of all, the sampling size num_i can be calculated by:

$$num_i = max\Big(rate_i \times len_i, min\Big(\frac{r \times \varphi}{m}, len_i\Big)\Big), \qquad (1)$$

where $rate_i$ represents the sampling rate, and φ is defined as the lower bound of the sampling size. r and m denote

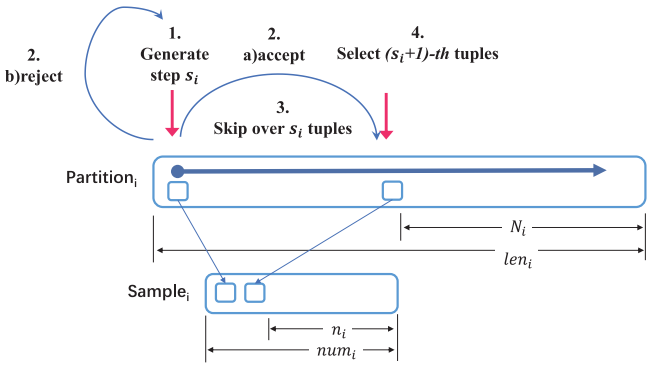


Fig. 3. The process of sampling.

the number of reducers and mappers in a shuffle phase, respectively.

The two most critical parts are the step generation and the decision to accept the step. We select the first provided choice parameters which are proposed by Vitter [32], which can complete the sampling in linear complexity on the average with a reliable accuracy. Thus, the step s_i can be generated by:

$$s_i = \lfloor x \rfloor = \left\lfloor N_i\left(1 - u^{1/n_i}\right)\right\rfloor, \tag{2}$$

where u is uniformly distributed on the unit interval. For Step (2), there are two relevant approaches to judge whether to accept the step. These two methods are applied to different periods of sampling. First, at the beginning of sampling, the decision can be generated by Eqs. (3) and (4):

$$1 - \frac{x_i}{N_i} \leq \left[\frac{1}{v}\prod_{k=0}^{s_i}\left(1 - \frac{n_i - 1}{N_i - k}\right)\right]^{\frac{1}{n_i - 1}} \tag{3}$$

$$\left(\frac{vN_i}{N_i - n_i + 1}\right)^{\frac{1}{n_i - 1}} \leq \left(1 - \frac{s_i}{n_i - 1}\right)\frac{N_i}{N_i - x_i}. \tag{4}$$

If the Eqs. (3) and (4) are satisfied, step s_i can be accepted. The variable v is a random decimal, which is uniformly distributed on the unit interval.

Since this above method becomes slower when n_i is close to N_i. For this situation, we can set a threshold α in this model. When $n_i/\alpha \leq N_i$, the other simpler approach is selected to complete the remaining sampling. If Eq. (5) is met, the step s_i can be accepted.

$$1 - v \geq \prod_{k=0}^{s_i}\left(1 - \frac{n_i}{N_i - k}\right). \tag{5}$$

The sampling procedure can be finished until $n_i = 0$. To avoid memory overflow and improve the efficiency, the key clusters during the sampling are combined in parallel. In other words, for ith partition, the frequency $freq_{i,k}$ of a key k would be updated once a tuple of key k is added to the samples. Therefore, we can obtain the frequency list of keys in each partition after sampling.

Considering that some operations need to perform map side combination, it is not appropriate to simply represent the key distribution of intermediate data with that of RDD tuples. Hence, a function $trans(k, i)$ is designed to denote the number of tuples with a key k in the ith map task output in this model, which can be calculated as follows:

$$trans(i,k) = \begin{cases} \frac{freq_{i,k}}{rate_i}, & MSC = 0 \\ 1, & MSC = 1, \end{cases} \tag{6}$$

where $MSC = 1$ denotes that map side combination is performed. In this model, c_k denotes the number of all the tuples whose key is k, which is an estimation of the frequency of key k in the intermediate data, and can be calculated by:

$$c_k = \sum_{i=0}^{m-1}\frac{trans(i,k)}{ratio}. \tag{7}$$

Therefore, a general key distribution can be finally obtained. It is represented by C, and composed of all the c_k.

4.2 Data Sampling and Key Distribution Estimation Algorithm

The specific processes of data sampling and key distribution estimation are respectively provided in Algorithms 1 and 2 in detail.

Algorithm 1 adopts a step based rejection sampling method to process the data. It first generates a step s_i. When $n_i/\alpha < N_i$, it determines whether to accept the step according to the first approach by Eqs. (3) and (4). When $n_i/\alpha \geq N_i$, it makes the decision according to the second approach by Eq. (5). When a step is accepted, a key is added to the sample and its frequency is updated by $freq_{i,k} = freq_{i,k} + 1$. Thus, each element of the returned array $Sample$ including two part: the sampling rate and the frequency list of keys for a certain partition.

Algorithm 1. Step-based Rejection Sampling

Input:
Map partitions: MP.
Output:
The array of tuples including sampling rate and data: $Sample$.
$Sample = \varnothing$;
for each sampled partition i in map input data MP **do**
  k_{pos} denotes pos^{th} tuples in partition i;
  Initialize the sampling position, $pos = 0$;
  Initialize the frequency list for partition i, $freq_i = \varnothing$;
  Calculate real sample rate $rate_i$ and sampling size n_i;
  while $n_i > 0$ **do**
    Generate random decimal u and random step s_i;
    if $n_i/\alpha \leq N_i$ **then**
      Determine whether the first condition is satisfied;
    else
      Determine whether the second condition is satisfied;
    end if
    if $accept$ **then**
      $pos = pos + s_i + 1$;
      $//k$ denotes the pos^{th} key in partition i;
      $freq_{i,k} = freq_{i,k} + 1$;
      $N_i = N_i - 1 - s_i$;
      $n_i = n_i - 1$;
    end if
  end while
end for
$Sample = \bigcup_{i=0}^{m} Sample_i = \bigcup_{i=0}^{m}(rate_i, \bigcup freq_{i,k})$;
return $Sample$.

Second, as obtaining the sampling rate and key frequency list, Algorithm 2 merges the sampling results of all the partitions. $c_k = c_k + (trans(i,k)/ratio)$ is to add the weight of key k in the ith partition to the total weight c_k of key k. Therefore, we get a collection C as an estimation of the key distribution.

Algorithm 2. Key Distribution Estimation

Input:
The sampling result: $Sample$;
Output:
The collection of weights for key clusters: C.
for each sample result $Sample_i$ of ith partition **do**
  for each key k in $Sample_i$ **do**
    //Merge the key clusters in all the partitions;
    $c_k = c_k + (trans(i,k)/ratio)$;
  end for
end for
// Update the key clusters;
$C = \bigcup c_k$;
return C.

The time complexity of the intermediate keys distribution detection is $o(s \times (t+k))$, where s is the number of sampled partitions. t is the tuple number in each partition, and k is the number of distinct keys in each partition.

5 Partition Policy

5.1 Model of the Partition Algorithm

The standard of division is of quit significance for a partition method. Many previous studies divide data on account of the balance of intermediate data. A common measurement used by them for data skew is the coefficient of variation: $FoS = stddev(\vec{x})/mean(\vec{x})$, where x is a vector that contains the data size processed by each task. $stddev(x)$ denotes the Standard deviation of data size, and $mean(x)$ denotes the mean value of data size. In this equation, larger coefficient indicates heavier skew [17].

However, the balance of intermediate data can only guarantee the balance of executing shuffle operations, because the number of tuples often changes after a shuffle operation has been performed. For instance, for a self join operation, the number c of tuples changes to c^2. Therefore, not only we should take the balance of intermediate data into account, but also we must consider the change in the numbers of tuples after the shuffle operators.

Based on the ideas and perspectives of thinking mentioned above, this section formulates the integrated skew degree Isd as follows:

$$Isd = \frac{\sqrt{\frac{\sum_{j=0}^{p-1}\left(I_j - I_{avg} + \lambda(S(I_j) - S_{avg})\right)^2}{p}}}{I_{avg} + \lambda S_{avg}}, \tag{8}$$

where I_j denotes the ith partition of intermediate data and $S(I_j)$ is the tuple number of ith partition after shuffle operator. I_{avg} and S_{avg} is the corresponding average values. The variable λ is the ratio of normal-op workload to shuffle-op workload. As a matter of fact, Isd is the coefficient of variation about $I_j + \lambda S(I_j)$ that can be calculated as follows:

TABLE 2
Variable Declaration

$r, 0 \le j < r$	r: reducer number; j: one reducer;
$m, 0 \le i < m$	m: mapper number; i: one mapper;
C	the collection of the frequencies of key clusters;
W_j	the weight of splits j;
W_{avg}	the average weight of all the splits;
P_{redis}	the collection of the redistributive partition indices;
K_{redis}	the collection of the redistributive keys;
ST	the reassignment strategy for redistributive keys in K_{redis};
WB	the weight bounds for *range* partition.

$$I_j + \lambda S(I_j) = \sum_{k \in RP_j} c_k + \lambda \sum_{k \in RP_j} s(c_k) = \sum_{k \in RP_j} (c_k + \lambda s(c_k)), \tag{9}$$

where $k \in RP_j$ denotes that key k is assigned to jth reduce partition and $s(c_k)$ denotes the tuple number of key k after shuffle operator. Thus, when calculating the strategies, we should set the weight of each key as $c_k + \lambda s(c_k)$. For the sake of generality, we set $\lambda = 1$. On this basis, we can generate the actual partition strategies.

Two partition algorithms respectively based on the *hash* and *range* are proposed to achieve the data balancing. The range based method is designed for these applications which require a ordering result and the hash-based method is for the other general applications. For the jobs with the similar types of input, because the distributions of the intermediate keys would be consistent after the same RDD transformations, we just need to do the data sampling only once, and then the obtained partition policy can be reused by different jobs. For illustrative purposes, some significant variable declarations are presented in Table 2.

As illustrated in Fig. 4, the process of the hash-based key reassigning algorithm can be divided briefly into three steps:

1) Divide the sample collections into several splits by hash method.
2) Obtain the overload splits as redistributive partitions and select some of the keys in overload splits as the redistributive keys. Then, allocate these keys to other splits and record their relations.
3) Implement the key reassignment strategy in real shuffle phase.

For describing the hash-based key reassigning algorithm in detail, some specific data structures can be formalized as follows:

(1) W. A vector of length r whose element indicates the total weight of each split. For these key clusters c_k are divided in C, parameter W_j denotes the weight of the jth split. This algorithm first distributes these key clusters according to the hash method, and call each parts hash-splits. So W_j can be calculated by:

$$W_j = \sum_{c_k \in C} (c_k + s(c_k)) \times belong(k,j), \tag{10}$$

where the value of $belong(k,j)$ is set to 1 when $h_k = j$, otherwise, its value equals to 0. $h_k = (k.hashcode + r) \bmod r$. Therefore, W is an approximation of the intermediate data

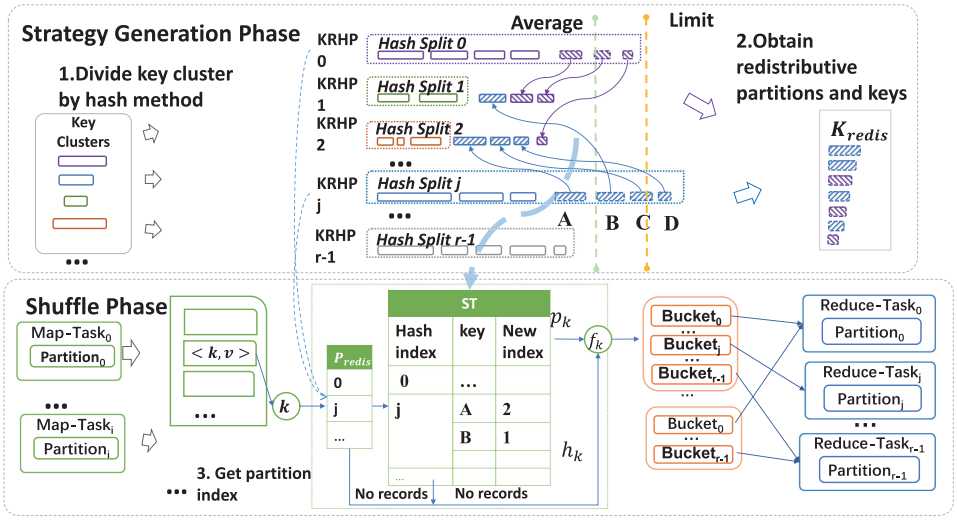


Fig. 4. The steps of hash-based key reassigning algorithm.

distribution. W_{avg} denotes the average weight of all the splits in W.

(2) P_{redis}. A collection that contains the indices of overload splits in W. Because the weight of an overload split is much greater than the average W_{avg}, P_{redis} is calculated by:

$$P_{redis} = \{j | W_j > W_{avg} \times tol\}, \tag{11}$$

where tol is used to adjust the allowed skew tolerance when filtering these partitions. Obviously, in the real shuffle phase, these reduce partitions whose indices are recorded in P_{redis} have much more intermediate data to process than the others do. Thus, some of their keys should be reassigned to the others for load balance.

(3) K_{redis}. An collection composed of the redistributive keys. When the jth hash-splits in W is regarded as overload, its keys can be divided into two parts: K^j_{hash} (They are keeped in the original hash-splits) and K^j_{redis} (They are reallocated to new splits). This model would try to put key with larger c_k into K^j_{hash} as more as possible. Meanwhile, it is better to make the weight of K^j_{hash} more close to the average weight W_{avg}. Accordingly, the K^j_{hash} and K^j_{redis} are supposed to satisfy:

$$W_{avg} - \sum_{k \in K^j_{hash}} c_k \leq min(K^j_{redis}), \tag{12}$$

where $min(K^j_{redis})$ indicates the key of the smallest c_k in K_{redis}. The weight of overload partition W_j should be updated as $\sum_{k \in K^j_{hash}}^{c_k}$. And the collection K_{redis} is the union of all the K^j_{redis}, which is shown as:

$$K_{redis} = \bigcup_{j \in P_{redis}} K^j_{redis}. \tag{13}$$

(5) ST. A collection that records the partition strategy for these reassigned keys. ST_j saves all reassignment information of keys in K^j_{redis}. This is actually a packing problem: how to assign all redistributive keys to boxes that are of different sizes because these partitions already contain some keys. In our works, we try the first fitting and best fitting algorithms, and the results show that the first fitting algorithm has greater performance, so that we set it as default.

After giving the specific definition of data structures, we consider it necessary to give a detailed explanation of Fig. 4. The area of a dashed box indicates the weight of a reduce split divided by the hash method. To decrease the skewness, the following steps are taken to solve this problem:

In the strategy generation phase, this model divides these key clusters into r splits by the hash method and saves the total weight of each split in W. Then, it can judge which splits exceed the preset limit and record them in P_{redis}. For each split in P_{redis}, such as 0, we try to select suitable keys to fill a space of W_{avg}, and put other keys into redistributive key set K_{redis}. Finally, these keys are allocated in K_{redis} to fill other small splits, and the reassignment information are recorded in ST. For instance, if key A in the primary jth hash-splits is allocated to the 2^{nd} reducer, a new index will be recorded in ST_j for key A.

In the real shuffle phase, when partitioning intermediate data, for each tuple, if the keys' hash index is recorded in P_{redis} and its new index is in K_{redis}, it can be allocated according to the new index p_k. If not, it will be allocated by the hash method. For each key k with the hash index is h_k, and its final index f_k can be calculated by:

$$f_k = \begin{cases} p_k, j = h_k \cap j \in P_{redis} \cap k \in K^j_{redis} \cap p_k \in ST_j \\ h_k, otherwise \end{cases}. \tag{14}$$

As illustrated in Fig. 5, the process of the range-based key splitting algorithm can be divided into three steps:

1) Sort the sampled key clusters by key and calculate the step that is used to divide the boundaries;
2) Select the boundary keys and calculate the weights of bounds;
3) Implement the weighted bounds in shuffle phase.

Range assigns the key cluster of a boundary key to a partition, but KSRP splits the key cluster and allocates them to different adjacent partition to make all the partitions have a similar size.

For the model of key clusters splitting algorithm based on *range*, some specific data structures can be formalized as follows:

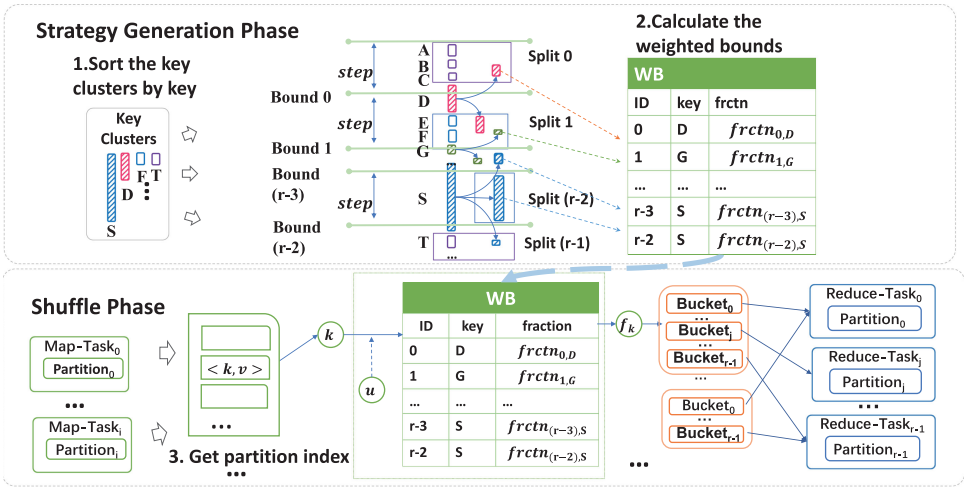


Fig. 5. The steps of splitting key clusters.

(1) *step*. A value denoting the interval to be used to define a bound. It is equal to the average weight of reduce partitions, calculated by:

$$step = \frac{1}{r}\sum^{l}(c_k + s(c_k)). \tag{15}$$

(2) *frctn*. A probability for a boundary key to be assigned to a certain reducer. We first sort the set C by the key k. Then the weights of key clusters can be accumulated as $curW_j$ in order. When the next key is k and $curW_j + c_k > step$, the key k is considered to be a boundary key. To distinguish a boundary key from ordinary keys, we use b_j to represent the jth boundary key and $c_{b_j} + s(c_{b_j})$ to represent the weight of its key cluster. Then we split the weight into two parts: $(step - curW_j)$ which is allocated to the jth partition and $(c_{b_j} + s(c_{b_j}) + curW_j - step)$ which is assigned to the new weight of key b_j for next step. So the probability for the boundary key b_j to allocate to the jth reducer is calculated as:

$$frctn_{j,k} = \frac{step - curW_j}{c_{b_j} + s(c_{b_j})} \tag{16}$$

(3) *WB*. A ordered vector that records the weighted boundary keys. WB is composed of $r-1$ boundary keys and relevant *frctn* as:

$$\begin{aligned} WB_j &= \left(b_j, frctn_{j,b_j}\right) \\ WB &= \left[WB_0, WB_1, \ldots, WB_j, WB_{r-2}\right]. \end{aligned} \tag{17}$$

where each WB_j in WB defines an upper bound of partition j.

After giving the specific definition of data structures, the detailed explanation of Fig. 5 is illustrated below.

In the first step, after the key cluster c_k sorted in C according to the key, we can select the boundary keys at intervals of *step*. To make sure that the weight of each split is equal to the average, each boundary key is split into several parts. There are several possible situations:

1) When $c_k + s(c_k) < step$, cluster of key k is split into two parts. For instant, in Fig. 5, boundary key A is split into two parts. The first part belongs to split 0, and the rest is allocated to split 1. So D becomes the upper bounds of partition 0, and the fraction of D is $frctn_{0,D}$.
2) When $c_k + s(c_k) > step$, cluster of key k might be split into more than two parts, which means that some bounds might be the same key. In Fig. 5, the cluster of key S are split into three parts, so key S are the boundary keys of two bounds. If the fractions of these three parts are 20, 70 and 10 percent, the two bounds are $(S, 0.2)$ and $(S, 0.7)$.

During the shuffle phase, for a given key k, the value of WB can be calculated to find the first boundary key which is not less than key k in the specified order. If the jth boundary key is greater than the key k, the cluster of k will be allocated to the jth reducer. If it is equal to k, this module will generate a random decimal u. Finally, it sums up *frctn* of bounds with a same key k in sequence until the total is greater than u. In other words, the purpose is to find a value of θ to satisfy the conditions:

$$\sum_{j=a}^{\theta} frctn_{j,b_j} \geq u, \xi_1 \leq \theta \leq \xi_2, \tag{18}$$

where ξ_1 and ξ_2 denote the indices of the first key and the last key in WB, which are equal to key k. The θ denotes the first key, which meets Eq. (18).

If θ exists, the key k will be assigned to reducer θ. Otherwise, it means that there are approximately $(1 - \sum_{j=\xi_1}^{\xi_2} frctn_{j,k})$ of the tuples left in partition $(\xi_2 + 1)$, and the key would be allocated to the reducer $(\xi_2 + 1)$.

Therefore, during the shuffle phase, we can dispatch these keys on the basis of their fractions for different reducers. This process of getting the partition index can be described as follows:

$$f_k = \begin{cases} j, & b_{j-1} < k < b_j \\ \theta, & b = k \cap \exists \sum_{j=\xi_1}^{\theta} frctn_{j,b_j} \geq u \\ \xi_2 + 1, & b = k \cap \sum_{j=\xi_1}^{\xi_2} frctn_{j,b_j} < u \end{cases}. \tag{19}$$

TABLE 3
Examples for Getting Partition Index

ID	Bounds	Key = F	Key = S	Key = S
0	(D, 0.35)	$F > D$;	$S > D$;	$S > D$;
1	(G, 0.45)	$F < G$; accept	$S > G$;	$S > G$;
...	...	...	...	...
r-3	(S, 0.2)		$S = S$; set $u = 0.85$; $ratio = 0.2$; $ratio < u$;	$S = S$; set $u = 0.95$; $ratio = 0.2$; $ratio < u$;
r-2	(S, 0.7)		$ratio = 0.9$; $ratio > u$; accept	$ratio = 0.9$; $ratio < u$; accept

The example in Fig. 5 shows some typical cases of getting partition index in Table 3. If the current key is F, it will be assigned to the reducer because $b_0 = D < F < b_1 = G$. If the current key is S and random decimal u is 0.85, its partition index is $r-2$ because $\sum_{j=r-3}^{r-2} frctn_{j,S} = frctn_{r-3,S} + frctn_{r-2,S} = 0.2 + 0.7 = 0.9 > 0.85$. But when u is 0.95, its index is $r-1$ because $\sum_{j=r-3}^{r-2} frctn_{j,S} = 0.9 < 0.95$.

5.2 Key Cluster Reassigning Partition Algorithm Based on Hash

The hash-based algorithm attempts to generate a key reassigning strategy. Algorithm 3 is used to obtain redistributive partitions and keys.

Algorithm 3. Obtaining Reassigned Reduce Partitions and Keys

Input:
The array of key cluster: C;
Output:
Redistributive partition indices and keys: (P_{redis}, K_{redis}).
Calculate the weights $W = \{W_j\}$ of splits divided by hash method;
Calculate the average weight W_{avg};
for $j = 0$ to $r-1$ **do**
  if $W_j > W_{avg} \times tol$ **then**
    // Get the overload split j;
    $P_{redis} = P_{redis} \cup j$;
    $rest = W_{avg}$;
    Sort elements in K^j_{all} by c_k;
    for each $k \in j$th hash splits **do**
      if $rest \geq c_k$ **then**
        $rest = rest - c_k - s(c_k)$;
      else
        // Get the redistributive key k;
        $K^j_{redis} = K^j_{redis} \cup k$;
      end if
    end for
    $W_j = W_{avg} - rest$;
  end if
end for
return P_{redis}, $\bigcup_{j=0}^{r-1} K^j_{redis}$.

Algorithm 3 first divides the key clusters in C by hash method, and then for each split it records the keys and total weight W_j. When $W_j > W_{avg} \times tol$, the jth split is overload. And the value j is then recorded in the P_{redis}, as one of the redistributive indices. This algorithm sorts the key in each redistributive partition by the value of $c_k + s(c_k)$ in a descending order and set the remaining space $rest$ to W_{avg}. Sequentially, if $c_k + s(c_k) > rest$, this step adds k into K_{redis}.

As obtained the W_j and K_{redis} after Algorithm 3, Algorithm 4 is designed to work out an appropriate strategy to dispatch these redistributive keys in K_{redis}. The keys in K_{redis} are reassigned using first fitting algorithm. And meanwhile, it is necessary to record the partition strategy consisting of P_{redis} and ST, which will be implemented in the shuffle phase. The output of this algorithm is a collection that records the partition strategy for these reassigned keys.

Algorithm 4. Dispatching Redistributive Keys

Input:
The collection of redistributive keys: K_{redis};
The collection of split weights: $W = \{W_0, W_1, \ldots, W_{p-1}\}$.
Output:
The key reassignment strategy, ST.
Sort the W in ascending order;
//d denotes the order index and p_k denotes the reduce index;
for $d = 0$ to $r-1$ **do**
  $rest = W_{avg} - W_d$;
  for each key $k \in K_{redis}$ **do**
    if $c_k < rest$ **then**
      $rest = rest - c_k - s(c_k)$;
      $j = getHashPartition(k)$;
      //Reassign key k to reducer p_k;
      //Record this relation in ST_j;
      $K_{redis} = K_{redis} - k$;
    end if
  end for
  $W_d = W_d - rest$;
end for
Assign larger remaining key cluster to smaller reduce partition;
return ST.

5.3 Key Cluster Splitting Partition Algorithm Based on Range

For the key cluster splitting partition algorithm based on *range*, it is significant to identify the weighted bounds. Algorithm 5 illustrates this process in detail.

In the first place, it sorts the key cluster collection C and calculates the the average weight $step$. The parameter $rest$ which denotes the remainder space of a partition is initialized as $step$. When the weight of key cluster is greater than $rest$, it means that the key is selected as

a bound between two neighbouring partitions. This algorithm calculates the $frac_{i,k}$ and sets the weight as $c_k + s(c_k) - rest$ which should be assigned to the next partitions. After adding $< k, frac_{i,t} >$ to the ordered collection WB, we can continue to figure out the next bounds. After working out all the bounds or traversing all the key cluster, the process terminates.

Algorithm 5. Splitting Key Cluster and Partitioning By Range

Input:
The array of key cluster: C;
The number of reduce partitions: r.
Output:
The collection of weighted range bounds: WB.
Initialize the collection of bounds, $WB = \varnothing$;
Initialize the index of tuple in KW, $t = 0$;
Initialize the index of partition, $j = 0$;
Sort the C by key;
Calculate the total weight W_{total} of all the tuples in KW;
$step = W_{total}/r$;
while $j < r - 1$ **do**
  $rest = step$;
  if $rest \leq c_k + s(c_k)$ **then**
    $c_k = c_k + s(c_k) - rest$;
    $frac_{j,k} = rest/all_k$;
    $WB_j = (k, frac_{j,k})$;
    Start to calculate next partition, $j = j + 1$;
    $rest = step$;
  else
    $rest = rest - c_k - s(c_k)$;
    Get next key k and set all_k to $c_k + s(c_k)$;
  end if
end while
return WB.

5.4 Key Cluster Allocation Strategy

In the shuffle phase, when dividing intermediate data, Spark determines which reducer a tuple belongs to in accordance with a function $getPartition$ based on the key.

For the hash-based key reassigning algorithm, according to the information recorded in P_{redis} and K_{redis}, Algorithm 6 obtains the final reduce index of key k. First, it calculates the hash index j. Only when the jth partition is redistributive, this algorithm traverses ST_j and attempts to find the real reducer index p_k. In this way, the time cost of obtaining the partition can be extremely decreased. Besides, key k is allocated according to hash method if j is not a distributive index.

For the range-based key cluster splitting algorithm, the weighted bound WB has been calculated using Algorithm 5. On this basis, Algorithm 7 is used to determine the reduce index of a certain key k by searching the bounds in WB. First, it searches for a boundary key b_j that is not less than k. If $b_j > k$, k is allocated to jth reducer. However, if $b_j = k$, we can generate a random decimal u and sum up the fraction of bounds equal to key k until the sum is greater than the u. The index of the last added bound is the final index of key k.

Algorithm 6. Getting Partition Based On Key Reassigning

Input:
The key reassigned policy: ST;
The key of the tuple: k;
The collection of reassigned partition index: P_{redis}.
Output:
The final partition k belongs to: p_{final}.
//Calculates the hash index j;
//Attempts to find the real reducer index p_k;
$j = getHashPartition(k)$;
if $j \in P_{redis}$ **then**
  if $p_k \in ST_j$ **then**
    $f_k = p_k$;
  else
    $f_k = h_k$;
  end if
end if
return f_k.

Algorithm 7. Getting Partition Based On Key Cluster Splitting

Input:
The key: k;
The weighted bounds: WB.
Output:
The final partition k belongs to: f_k.
Initialize partition index $j = 0$;
// b_j denotes the jth boundary keys;
while $j \leq r$ and $k > b_j$ **do**
  $j = j + 1$;
end while
//Only when k is one of the boundary keys, we execute the following code to calculate its final index;
if $j \leq r - 1$ and $k = b_j$ **then**
  $ratio = frctn_{j,k}$;
  Generate a random decimal u;
  while $j \leq r$ and $k = b_j$ and $u > ratio$ **do**
    $j = j + 1$;
    if $j < r$ **then**
      $ratio = ratio + frctn_{j,k}$;
    end if
  end while
end if
$f_k = j$;
return f_k.

6 Experiments

6.1 Experiment Setting

In this section, SKSRP is evaluated on a practical cluster based on Spark 2.2.0, which includes 8 worker nodes and 1 master. The hardware and software configurations are shown in Table 4. We have supplemented this partition algorithm to the native method *defaultPartitioner* in Spark-core source project. Hence, the Spark progress can use our achievement through invoking our $getPartition$ method in SKRSP algorithm. (Our codes are to be shared at the open-source code repository, GitHub, after the manuscript is accepted.)

TABLE 4
Configuration of Cluster

Type	Configuration
Environment	Ubuntu 12.04, JDK 1.8, Hadoop 2.6.0, Spark 2.2.0
CPU	4 Cores, 2.7 GHz
Memory	28G

For evaluating the effect, as shown in Table 5, one application requiring map side combination (*WordCount*) and three types of application exhibiting significant data skew (*Join*, *Sort*, *PageRank*) are selected. The application *WordCount* is used to test the the influence of map side combination on intermediate data distribution estimation. And the rest three applications are used to evaluate the effect of SKRSP on skew mitigation.

In our experiments, the following partition methods are chosen for comparison.

DefHash: (*Default Hash Partition* [15]). In the original MapReduce Frameworks, such as Hadoop and Spark, it is a default hash partition method for almost all applications except for sort, which can partition intermediate data extremely fast but easily makes the data distribution unbalanced over reduce partitions.

DefRange: (*Default Range Partition* [16]). It is another default partition method used in traditional MapReduce Frameworks for relative sort applications. Unfortunately, when there are keys with fewer types but large numbers, the range method stands a good chance to cause data skew.

SP-P: (*SP-Partitioner* [21]). It is a partition method designed for Spark that makes each bucket which collects the data from the same map task be assigned equal sized data. It partitions the intermediate data on the basis of the tuple numbers.

6.2 Performance Results

6.2.1 Sampling Method Evaluation

To evaluate the accuracy of our sampling algorithm, we conduct a set of experiments to compare our method with the Spark random sampler and the sampling method used by *range* partitioner. We generate average 3000,000 strings for each partition following Zipf distributions [33] with parameter $\sigma = 1.0$. Considering the data distribution of map partitions is not always uniform, four types of data sets are generated to deal with this situation: a) uniform partitions with 5000 distinct keys; b) uniform partitions with 50000 distinct keys; c) skew partitions with 5000 distinct keys; d) skew partitions with 50000 distinct keys. Because the *range* sampler has no key combination during sampling for each partition, it causes out of memory when the sampling rate is slightly larger. So the sampling rate is set at 3.3 percent which is constrained by the amount of memory. However, the experiments are still interested about the effect of range sampler with larger sampling rate. And we implement the key combination on this case and conduct experiment on sampling rate of 20 percent. To give a rough idea of the accuracy of these sampling methods, the root mean square error is calculated for all distinct keys, which can be calculated by:

$$acc = \sqrt{\frac{\sum_{c_k \in C}(c_k - c_k^{real})^2}{|C|}}. \tag{20}$$

The acc values for our method, *range* sampler and Spark random sampler are shown in Table 6. It is obvious that the accuracy of our method is much more stable than other methods. To compare with the sampling method in LIBRA [17], this experiment generates data with 65535 distinct keys. We work out the root mean square error is 70, which is smaller than 183 of LIBRA.

There are several control parameters in our method: $ratio$ is the proportion of the sampled parts in all map partitions, sampling rate $rate$ for each partition, the lowest bound φ of the sampling size and sampling threshold α. For the later two parameters, we set $\varphi = 100$ and $\alpha = 0.07$. $ratio$ is set as $8/m$ when there are only one shuffle phase, otherwise for multiple shuffle phases, it is set as 1.0.

For the parameter of sampling rate, considering its potential impact on partition result, this work conducts a set of experiments to observe the variance of FoS over different sampling rate. The data sets are generated following the Zipf distributions ($\sigma = 0.8$) and the distinct key numbers are set to there levels: 10^4, 10^5 and 10^6. We select $partitionBy$ for test, because it has no map side combination and no extra computation but just divide the data into several new partitions, which is suitable for observing the effects of sampling results on the partition. In this experiment, the data sets are about 8.8G. And the overhead of sampling and generating a strategy is less than 18s. The final experiment results are showed in Table. 7, which indicates that for data with fewer distinct keys, a small sampling rate has already achieved good partition balance, but for data with more distinct keys, the rate is supposed to be larger.

TABLE 5
Benchmark Types

Application types	Benchmarks
map side combination application	WordCount
distributed database operations	Join
simple applications	Sort
search engine applications	PageRank

TABLE 6
Accuracy Comparison

sampler	rate	U 5,000	U 50,000	S 5,000	S 50,000
skrsp	3.3%	1311	296	581	207
range	3.3%	1622	460	883	245
random	3.3%	1897	361	743	218
skrsp	20%	288	110	232	78
range	20%	538	171	276	87
random	20%	302	111	210	71

6.2.2 Key Distribution Estimation and Partition Strategy Generation Testing

In the hash based key reassigning partition algorithm, these are two important points been considered: 1) the influence of map side combination on intermediate data distribution

TABLE 7
Sampling Experiments

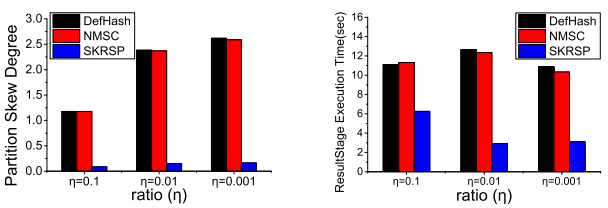
key number	sampling rate	sampling time(s)	strategy generation time(ms)	*sidd*	reduce time(s)
10^4	0	0	0	0.748	148
10^4	1×10^{-5}	15	47	0.512	84
10^4	5×10^{-5}	16	49	0.331	78
10^4	1×10^{-4}	16	48	0.283	63
10^4	5×10^{-4}	16	51	0.298	58
10^4	5×10^{-2}	17	71	0.296	64
10^5	0	0	0	0.748	95
10^5	1×10^{-4}	14	67	0.532	81
10^5	5×10^{-4}	16	144	0.323	72
10^5	1×10^{-3}	16	180	0.262	67
10^5	5×10^{-3}	16	242	0.300	62
10^5	1×10^{-2}	17	218	0.302	67
10^6	0	0	0	0.748	88
10^6	1×10^{-3}	12	207	0.512	82
10^6	1×10^{-2}	13	748	0.331	72
10^6	5×10^{-2}	14	2272	0.283	56
10^6	1×10^{-1}	15	2586	0.298	68
10^4	2×10^{-1}	16	1895	0.296	60

estimation and 2) the influence of intermediate data balance and RDD partition balance on execution time. To illustrate the function of these two point, two sets of experiments are designed on these data sets with different data features which easily causes data skew. There are two types of data features: a) key skew (existing keys with extremely high frequencies), b) hash skew (many different keys whose hash codes point to the same reduce index).

To test the first point, a group of experiments are evaluated by running *WordCount* benchmarks on data set with hash skew. This is because *WordCount* needs map side combination, and it is suitable to testing the influence to the forecast of the key distributions from the map side combination. And hash skew is the main cause of its performance degradation.

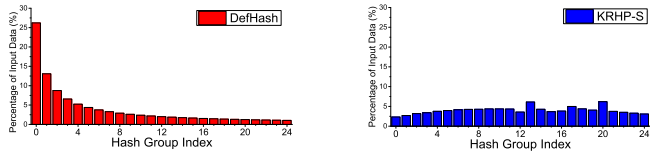
In these experiments, the numbers of mappers and reducers are set to 32 and 8 respectively. Then we make RDD with 32 partitions and generate 10^7 key/value tuples for each partition. For all the tuples, there are 10^5 keys whose hash index are equal to 0 and $7\eta \times 10^5$ keys whose hash indices are uniformly distributed from 1 to 7. Good key distribution approximation can achieve better intermediate partition balance, so we can indirectly observe the effect of estimation through the metric *FoS*.

Fig. 6 shows us the experiment results. As the results of NMSC shows, because the intermediate data have been combined in the map-side processing, and the distributions



(a) Partition skew degree (b) ResultStage Execution time

Fig. 6. The effect of mapSideCombine.



(a) DefHash (b) KRHP

Fig. 7. Key distribution when $\rho = 1.0$.

of the intermediate keys have been changed. Hence, the key distributions in later processes are not the same with map process. If the effect of map side combination is not included, the partition method might obtain the fake information and cannot detect the occurrence of data skew.

For the second point, the reason why we choose *Join* is because it is a typical data operation with higher likelihood of data skew. Data skew has a great impact on the performance of *Join* operation. And the size variation of the data in the RDD before and after Join transformation is also very large. To run the join benchmark, the experimental data sets are generated by dividing it into several groups according to key's hash code. Each group contains approximately 1000 keys whose numbers are very close. The number of keys in each group is distinct, so that we can simulate the situation that both hash skew and key skew exist in the input data.

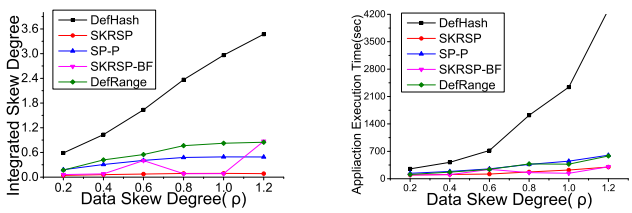
The numbers of the keys obey the Zipf distribution. We use a zipf sampler whose skew degree is ρ to sample these id arrays of all hash groups. Once obtain an id, a corresponding key of the hash group would be generated. The parameter ρ is used to control the skew degree: a larger ρ indicates a higher skew degree. In this experiment, the way of inner join is adopted. The type of key is integer, and the type of value is a byte array of length 10.

If there are 25 groups and 25 partitions, the distribution of keys on *DefHash* can be illustrated as shown in Fig. 7a. Fig. 7b presents the allocation result if this input data is partitioned by *SKRSP*, which is much more uniform than that in Fig. 7a.

In these experiments, we set up two test cases:

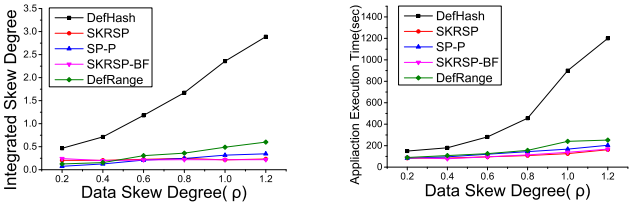
a) a large RDD joins a large RDD ($580\text{M} \times 580\text{M}$);
b) a large RDD joins a small RDD ($4\text{G} \times 18\text{M}$).

Figs. 8 and 9 show the *Isd* and application execution time of these two cases when σ varies from 0.2 to 1.2. When ρ increases, for DefHash, both the *Isd* and the application execution times grow rapidly; at the same time, for *SKRSP* and *SP-P* these indicators are maintained in a stable state with considerably low values. Indeed, when running *DefHash*, we discovered that a reduce task slows down the execution of the entire application because it has to process many



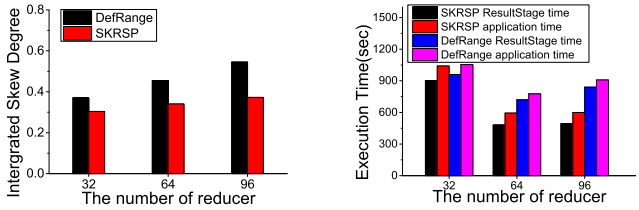
(a) Integrated Skew Degree (b) Application Execution Time

Fig. 8. Performance of Join when ρ varies (580 MB $\times$ 580 MB).



(a) Integrated Skew Degree (b) Application Execution Time

Fig. 9. Performance of join when ρ varies (4 GB×18 MB).



(a) Integrated skew degree (b) Execution time

Fig. 11. Sort benchmarks on primary data set.

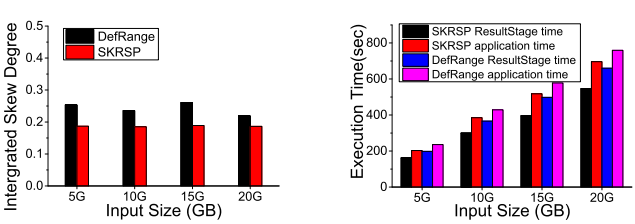
more tuples than the others do. *SKRSP* and *SP-P* effectively avoids the occurrence of this situation so that it performs much faster than *DefHash*. However, *SKRSP* performs significantly better than *SP-P*, and *SKRSP* using first fitting is more stable than *SKRSP-BF* using best fitting. Therefore, we use the first fitting algorithm as the default method.

6.2.3 BigDataBenchmark Testing

In this section, two benchmarks of BigDataBench are selected to test our method: simple application *Sort*, search engine application *PageRank*. Both of them exhibit significant data skew in real world. *Sort* is a typical operation which need products the ordered output, which is suitable to comparing the partition effects of the algorithms KSRP and *Range*. And *Pagerank* is a widely used algorithm in most search engine applications. Because it is very easy to cause data skew when hot data occurs, *Pagerank* is very suitable to observing the effect of performance improvement based on SKRSP.

For the *Sort* benchmark, the scalable data sets of Wikipedia entries are produced by text generator of BDGS. There are four sizes of data: 5G, 10G, 15G and 20G, and the parallelism is set as 64. Fig. 10 records the experimental results which compared *SKRSP* with the *default range partition* in Spark, including integrated skew degree, ResultStage execution time and application execution time.

Fig. 10a shows that the *SKRSP* can maintain a lower integrated skew degree. Fig. 10b indicates that the ResultStage execution time accounts for a large proportion of the application execution time and *SKRSP* can reduce the time of ResultStage so that the overall time is decreased by at least 10 percent. We also run the *Sort* benchmark on primary Wikipedia dataset in different parallelism, 32, 64, 96. The results are shown in Fig. 11, which indicates that different parallelism causes distinct integrated skew degree. *SKRSP* can decrease the value of *Isd* and reduce the ResultStage execution time in each parallelism. But when the degree of skew reduction is small, the performance improvement is not remarkable.



(a) Integrated skew degree (b) Execution time

Fig. 10. Sort benchmarks on scala data sets.

The experiments adopt two kinds of data sets to test the benchmark *PageRank*: one primary Google web data set (web_Google) and two scala data sets by graph generator of BDGS (genGraph_20, genGraph_21). When the number of reducers is set as 32, this group of experiments compare *SKRSP* with *DefHash* and *SP-P*. The application execution time of each test is shown in Fig. 12. Compared to *DefHash*, both the *SKRSP* and *SP-P* can decrease the application execution time. However when the size of data set increase, *SKRSP* has a more prominent impact than *SP-P* because it takes into account both the balance of intermediate data and the balance of partitions after shuffle operator.

6.3 Experiment Summary

In this section, the performance of SKRSP is evaluated on some popular applications with both synthetic and real-world datasets. Through these results, we can find that:

- SKRSP step-based sampling method achieves a good approximation to the key distribution of the original dataset with little overhead.
- SKRSP takes the switch of map side combination into account, which can estimate key distribution more accurately.
- SKRSP can alleviate or avoid the partition skewness. Moreover, the more skewed the data sets, the more significant the effect of optimization. Note that not all application should use SKRSP.

7 Conclusion

In the Spark framework, the default partition method easily causes intermediate data distribution imbalance when the input and output for map task are skewed. This situation extends the task execution time. This paper attempts to extenuate skewness among reduce tasks. First, it implements a rapid step-based sampling job to obtain intermediate data, and then it takes map side combination into consideration to estimate the key distribution in intermediate data. Moreover,

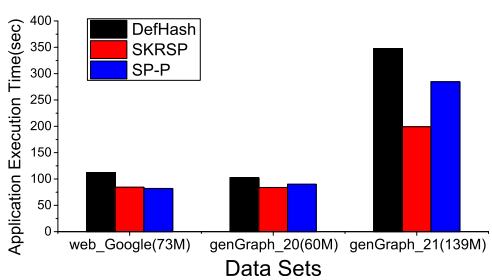


Fig. 12. Application execution times of PageRank.

it determines the appropriate partition strategy, which not only considers the balance of intermediate data distribution but also takes the partition balance after shuffle operator into account.

The experiments result shows that considering the influence of map side combination can estimate key distribution more acutely. And the experiments are based on three types of applications, which are prone to data skew, verify that the imbalance of data distribution among reduce tasks can be mitigated through this partition method. Moreover, with increasingly skewed data, the effect on partition skew mitigation becomes more remarkable.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (Grant Nos. 61572176, L1624040, L182400035, 61873090), the National Key Research and Development Program of China (2017YFB0202201).

References

[1] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proc. Usenix Conf. Hot Topics Cloud Comput.*, 2010, pp. 10–10.

[2] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. Mccauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. Usenix Conf. Netw. Syst. Des. Implementation*, 2012, pp. 2–2.

[3] Y. Le, J. Liu, F. Ergun, and D. Wang, "Online load balancing for mapreduce with skewed data input," in *Proc. IEEE Conf. Comput. Commun.*, 2014, pp. 2004–2012.

[4] "Hadoop," (2014, May). [Online]. Available: http://hadoop.apache.org

[5] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. N. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters*," in *Proc. USENIX Conf. USENIX Annu. Tech. Conf.*, 2014, pp. 1–12.

[6] P. Beame, P. Koutris, and D. Suciu, "Skew in parallel query processing," in *Proc. 33rd ACM SIGMOD-SIGACT-SIGART Symp. Principles Database Syst.*, 2014, pp. 212–223.

[7] B. Palanisamy, "Purlieus: Locality-aware resource allocation for mapreduce in a cloud," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2011, pp. 1–11.

[8] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for mapreduce," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, 2011, pp. 570–576.

[9] J. Tan, S. Meng, X. Meng, and L. Zhang, "Improving reducetask data locality for sequential mapreduce jobs," in *Proc. IEEE INFOCOM*, 2013, pp. 1627–1635.

[10] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 17–24.

[11] P. Dhawalia, S. Kailasam, and D. Janakiram, "Chisel: A resource savvy approach for handling skew in mapreduce applications," in *Proc. IEEE 6th Int. Conf. Cloud Comput.*, 2013, pp. 652–660.

[12] J. Polo, C. Castillo, D. Carrera, Y. Becerra, I. Whalley, M. Steinder, J. Torres, and E. Ayguad, "Resource-aware adaptive scheduling for mapreduce clusters," in *Proc. Int. Middleware Conf.*, 2011, pp. 180–199.

[13] J. Tan, X. Meng, and L. Zhang, "Coupling task progress for mapreduce resource-aware scheduling," in *Proc. IEEE INFOCOM*, 2013, pp. 1618–1626.

[14] K. H. Lee, Y. J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel data processing with mapreduce: A survey," *ACM Sigmod Rec.*, vol. 40, no. 4, pp. 11–20, 2012.

[15] "Hashpartitioner," (2015, Mar.). [Online]. Available: http://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.HashPartitioner

[16] "Rangepartitioner," (2015, Mar.). [Online]. Available: http://spark.apache. org/docs/latest/api/scala/index.html#org.apache.spark.RangePartitioner

[17] Q. Chen, J. Yao, and Z. Xiao, "Libra: Lightweight data skew mitigation in mapreduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2520–2533, Sep. 2015.

[18] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 17–24.

[19] Y. C. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2012, pp. 25–36.

[20] Z. Tang, X. Zhang, K. Li, and K. Li, "An intermediate data placement algorithm for load balancing in spark computing environment," *Future Generation Comput. Syst.*, vol. 78, pp. 287–301, 2016.

[21] G. Liu, X. Zhu, J. Wang, D. Guo, W. Bao, and H. Guo, "SP-Partitioner: A novel partition method to handle intermediate data skew in spark streaming," *Future Generation Comput. Syst.*, vol. 86, pp. 1054–1063, 2017.

[22] Y. Xu and P. Kostamaa, "Efficient outer join data skew handling in parallel dbms," *Proc. VLDB Endow.*, vol. 2, no. 2, pp. 1390–1396, Aug. 2009. [Online]. Available: https://doi.org/10.14778/1687553.1687565

[23] Y. Xu, P. Kostamaa, X. Zhou, and L. Chen, "Handling data skew in parallel joins in shared-nothing systems," in *Proc. ACM SIGMOD Int. Conf. Manag. Data*, 2008, pp. 1043–1052.

[24] Y. Xu and P. Kostamaa, "A new algorithm for small-large table outer joins in parallel DBMS," in *Proc. IEEE Int. Conf. Data Eng.*, 2010, pp. 1018–1024.

[25] E. Ardizzoni, A. A. Bertossi, M. C. Pinotti, S. Ramaprasad, R. Rizzi, and M. V. S. Shashanka, "Optimal skewed data allocation on multiple channels with flat broadcast per channel," *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 558–572, May 2005.

[26] J. W. Stamos and H. C. Young, "A symmetric fragment and replicate algorithm for distributed joinsyout," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1345–1354, Dec. 1993.

[27] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Handling data skew in mapreduce," in *Proc. Int. Conf. Cloud Comput. Serv. Sci.*, 2012, pp. 574–583.

[28] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in mapreduce based on scalable cardinality estimates," in *Proc. IEEE Int. Conf. Data Eng.*, 2012, pp. 522–533.

[29] L. Cheng and S. Kotoulas, "Efficient skew handling for outer joins in a cloud computing environment," *IEEE Trans. Cloud Comput.*, vol. 6, no. 2, pp. 558–571, Apr.-Jun. 2018.

[30] L. Cheng, S. Kotoulas, T. E. Ward, and G. Theodoropoulos, "Efficiently handling skew in outer joins on distributed systems," in *Proc. IEEE/ACM Int. Symp. Cluster Cloud Grid Comput.*, 2014, pp. 295–304.

[31] J. Yu, H. Chen, and F. Hu, "Sasm: Improving spark performance with adaptive skew mitigation," in *Proc. IEEE Int. Conf. Progress Informat. Comput.*, 2016, pp. 102–107.

[32] J. S. Vitter, "Faster methods for random sampling," *Commun. ACM*, vol. 27, no. 7, pp. 703–718, Jul. 1984. [Online]. Available: http://doi.acm.org/10.1145/358105.893

[33] J. Lin, et al., "The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce," in *Proc. 7th Workshop Large-Scale Distrib. Syst. Inf. Retrieval*, 2012, pp. 2000–2009.

[34] Z. Tang, W. Ma, K. Li, and K. Li, "A data skew oriented reduce placement algorithm based on sampling," *IEEE Trans. Cloud Comput.*, (2016, Sept.). [Online]. Available: https://doi.org/10.1109/TCC.2016.2607738

[35] J. Chen, K. Li, Z. Tang, K. Bilal, S. Yu, C. Weng, and K. Li, "A parallel random forest algorithm for big data in a spark cloud computing environment," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 4, pp. 919–933, Apr. 2017.

[36] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in mapreduce based on scalable cardinality estimates," in *Proc. IEEE Int. Conf. Data Eng.*, 2012, pp. 522–533.

Zhuo Tang received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2008. He is currently an associate professor of the College of Computer Science and Electronic Engineering at Hunan University, and is the associate chair of the Department of Computing Science. His majors are distributed system, cloud computing, and parallel processing for big data, including distributed machine learning, security model, parallel algorithms, and resources scheduling and management in these areas. He is now a core member in the open source community of OpenStack. He has published almost 60 journal articles and book chapters such as the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Information Forensics and Security*, FGCS, etc. He is a member of ACM and CCF.

Wei Lv received the bachelor's degree in computer science from Hunan Normal University, China. She is working towards the master degree at the College of Information Science and Engineering, Hunan University, China. Her research interests include the parallel computing, big data parallel processing, distributed system architecture, especially the improvement and optimization of MapReduce framework.

Kenli Li received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar with the University of Illinois at Urbana-Champaign from 2004 to 2005. He is currently a full professor of computer science and technology at Hunan University, the dean of the College of Information Sciences and Engineering of Hunan University, and the deputy director in the National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, and grid and cloud computing. He has published more than 160 research papers in international conferences and journals such as the *IEEE Transactions on Computers*, the *IEEE Transactions on Parallel and Distributed Systems*, JPDC, ICPP, ICDCS, etc. He is an outstanding member of CCF. He is a senior member of the IEEE and serves on the editorial board of the *IEEE Transactions on Computers*.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published almost 500 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, the *IEEE Transactions on Services Computing*. He is a fellow of the IEEE.