Parallel Computing 100 (2020) 102699

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

ImRP: A Predictive Partition Method for Data Skew Alleviation in Spark Streaming Environment

Zhongming Fu [a,b,c], Zhuo Tang [a,b,*], Li Yang [d], Kenli Li [a,b], Keqin Li [e]

[a] College of Information Science and Engineering, Hunan University, China
[b] National Supercomputing Center in Changsha, Hunan, China
[c] College of Computer Science and Technology, University of South China, China
[d] College of Computer and Communication Engineering, Changsha University of Science and Technology, China
[e] Department of Computer Science, State University of New York, New Paltz, New York, USA

ARTICLE INFO

Keywords:
MapReduce
Data skew
Load imbalance
Spark Streaming

ABSTRACT

Spark Streaming is an extension of the core Spark engine that enables scalable, high-throughput, fault-tolerant stream processing of live data streams. It treats stream as a series of deterministic batches and handles them as regular jobs. However, for a stream job responsible for a batch, data skew (i.e., the imbalance in the amount of data allocated to each reduce task), can degrade the job performance significantly because of load imbalance. In this paper, we propose an improved range partitioner (ImRP) to alleviate the reduce skew for stream jobs in Spark Streaming. Unlike previous work, ImRP does not require any pre-run sampling of input data and generates the data partition scheme based on the intermediate data distribution estimated by the previous batch processing, in which a prediction model EWMA (Exponentially Weighted Moving Average) is adopted. To lighten the data skew, ImRP presents a novel method of calculating the partition borders optimally, and a mechanism of splitting the border key clusters when the semantics of shuffle operators permit. Besides, ImRP considers the integrated partition size and heterogeneity of computing environments when balancing the load among reduce tasks appropriately. We implement ImRP in Spark-3.0 and evaluate its performance on four representative benchmarks: *wordCount, sort, pageRank*, and *LDA*. The results show that by mitigating the data skew, ImRP can decrease the execution time of stream jobs substantially compared with some other partition strategies, especially when the skew degree of input batch is serious.

1. Introduction

Currently, data stream processing and analysis have become increasingly urgent in many applications, such as ad-hoc queries, dynamic content delivery, and security event processing [1]. As an emerging stream processing framework, Spark Streaming [2] is built on top of Spark (a popular big data processing platform) [3] to support near real-time distributed stream processing. Instead of processing stream data one record at a time like traditional stream processing systems (e.g., Storm [4] and TelegraphCQ [5]), Spark Streaming processes many records together. These small sets of records are called micro-batches, and each of them is sent to Spark engine to be processed as a normal job.

A typical Spark job contains two types of stages namely map stage and reduce stage. Between them, the shuffle phase maintains the read/write relationship in which the key/value tuples outputted by map tasks are allocated to a certain partition and processed by relevant reduce tasks. However, when the data allocation among the partitions is imbalanced, it would lead to the load imbalance of reduce tasks because of various input sizes. This is what we call the data skew or reduce skew [6]. Data skew is one of the important performance bottlenecks of systems because the execution time of a stage may be delayed by the overloaded task.

Hash and *range* partitioner are two methods provided by Spark, which are responsible for assigning intermediate data according to their keys. The hash partitioner uses a simple hash function, whereas the range partitioner uses a set of partition borders to divide the key space of entire intermediate data. Unfortunately, both of them easily cause the reduce skew when the distribution of key/value tuples is non-uniform (some keys appear more popular than others). As an example, Fig. 1 shows the statistics of partial word frequencies among 10 batches when

* Corresponding author.
E-mail address: ztang@hnu.edu.cn (Z. Tang).

https://doi.org/10.1016/j.parco.2020.102699
Received 13 January 2020; Received in revised form 8 July 2020; Accepted 24 September 2020
Available online 2 October 2020

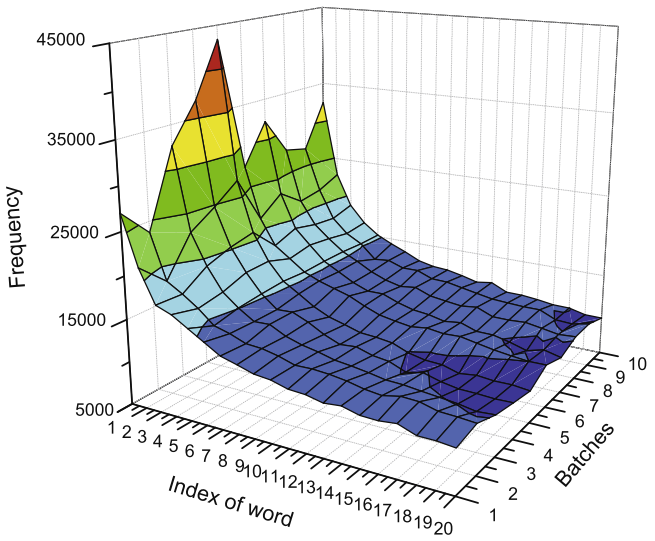


Fig. 1. Word frequency statistics of the top 20 words among 10 batches, each batch contains 3462 distinct words and about 9532980 records.

using the data set *Wikipedia Corpus* [7]. In fact, the word frequencies obey the *Zipf* distribution in the real world (a very common distribution of data generated by human society [8]). Therefore, when the words are regarded as the keys of the intermediate data by the jobs, the data is distributed non-uniformly and the reduce skew is prone to occur.

In Mapreduce-type frameworks, some useful partition methods were proposed by previous work to resolve the data skew problem [9], [10], [11], [12]. Most of them were designed for Hadoop and utilized the system characteristics. However, being different from the implementation of Hadoop [13], Spark adopts the multi-stage execution model and executes tasks stage by stage. Hence, the Hadoop-based approaches do not necessarily apply to Spark systems. In recent years, several researches optimized the partition methods of Spark [14], [15], but they are not suitable for Spark Streaming computing environments. To address this problem, in particular, Liu et al. [16] presented SP-partitioner to balance the partitions for stream jobs effectively. Nevertheless, the hash partition scheme cannot fit all types of jobs.

In the Spark engine, the shuffle operator (e.g., *reduceByKey*) in the reduce stage causes the shuffle phase and is first performed on the partitioned data [15]. However, only making the partition balance for the shuffle operation does not guarantee the load balancing of reduce tasks throughout their execution. Because the partition size may change after the shuffle operation. Moreover, when the computing environment in the real world is heterogeneous, the worker nodes can differ in the capabilities of processing the tasks. Hence, a good partition strategy should take these factors into consideration.

Motivated by the above work, this paper tries to relieve the data skew specially in Spark Streaming. In order to generate the key allocation strategy properly, it is necessary to know the distribution of intermediate data. But it is meaningless to count the key/value tuples after processing all input data. Fortunately, in most stream computing applications, the data characteristics do not change frequently [16]. As illustrated by Fig. 1, among the batches, the frequent keys always appear frequently, and vice versa. This paper proposes to use the past intermediate data of the previous batch processing to predict that of the forthcoming job, then through the optimized data partition scheme, the load of the reduce task can be balanced. The main contributions of this paper are summarized below.

- A new architecture is designed to mitigate the reduce skew in the shuffle phase of stream jobs, where the key distribution of the forthcoming job is predicted by the EWMA (*Exponentially Weighted Moving Average*) model. This step is separate from the normal job running and without extra delay.
- An improved range partitioner (ImRP) is proposed. Thr-ough comprehensively considering the partition balance before and after the shuffle operation, and by optimizing the calculation of partition borders, this partitioner can generate a more load-balanced partition scheme.
- When the performance of computing nodes is heterogeneous, ImRP can adjust its workload allocation accordingly; and when the semantics of the shuffle operator allow, ImRP can support the split of key cluster (all key/value tuples with the same key) to ensure the absolute equality.
- We implement our proposals in Spark-3.0 and evaluate its performance on several representative benchmarks. The experiment results show that ImRP can improve the execution efficiency of jobs by addressing the data skew successfully.

The rest of this paper is organized as follows. Section 2 surveys related work on data skew mitigation over Mapreduce frameworks. Section 3 introduces the system overview of ImRP. Section 4 presents the intermediate data prediction, and Section 5 describes the design of our partition method. Experiments and analysis are presented in Section 6. Section 7 concludes this paper.

2. Related Work

There are many real world applications exhibiting significant data skew, including parallel database operations (e.g., *Join* [17], *Group* [18], and *Aggregate* [19]), and search engine applications (e.g., *PageRank* and *Inverted Index*). In the past years, the problem of reduce skew has also been studied in MapReduce environments after the release of Hadoop [20], which is similar to our work:

Ibrahim et al. [21] proposed a novel algorithm called LEEN for locality-aware and fairness-aware key partitioning in MapReduce. It sorts all keys according to their fairness-locality values and uses a heuristic method to choose the node with the maximum fairness score for a reduce task to process the keys. Because the intermediate tuples are tracked after the map phase, LEEN embraces an asynchronous map and reduce scheme.

The main advantage of the work by Chen et al. [22] is LIBRA, a lightweight strategy that uses an innovative sampling method to ascertain the key distribution by sampling only a small fraction of the intermediate data. On the basis of the estimation, LIBRA generates the range partition scheme based on certain improvements, including chunk index for decreasing partition time and large cluster split.

In [23], Gufler et al. presented two load balancing approaches: fine partitioning and dynamic fragmentation. The former produces a fixed number of data partitions, and the latter dynamically splits large partitions into smaller portions and replicates data if necessary. Moreover, they define a new cost model to take into account non-linear reducer tasks.

Taking advantage of Hadoop's features, the above methods can obtain pretty good performance. However, as mentioned in section 1, they only balance the partitions of executing shuffle operation. While in Spark, one stage contains one or more RDD operations. Thus the Hadoop-based methods may not be competent for the work in Spark environments. At present, several partition strategies were proposed for Spark specifically.

Tang et al. [14] presented SCID, a splitting and combination algorithm that conducts a pre-run on the sample of input data before the normal job. Then by the statistics of keys, it forecasts the rough sizes of all clusters which will be produced for the whole input. At last, SCID puts the clusters into partitions and regards it as a bin-packing problem. Later, Tang et al. [15] further proposed SKRSP, a key reassigning and splitting partition algorithm. SKRSP designs two algorithms: hash based key reassigning algorithm and range based key splitting algorithm. Similar to [14], SKRSP uses a step-based algorithm for sampling the input data to obtain the general key distribution of intermediate data. Since the pre-run sampling job can postpone the main job, making them

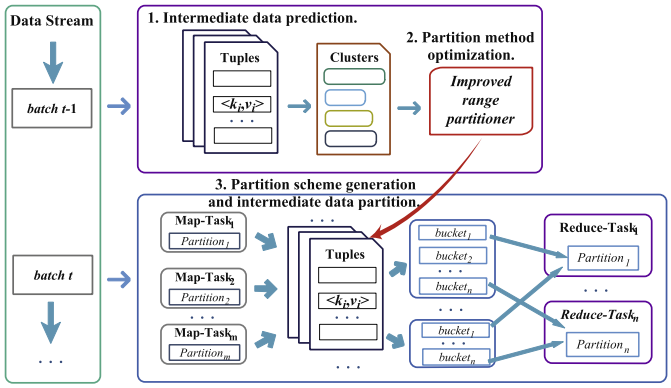


Fig. 2. The framework overview of ImRP.

is not a good choice for stream computing.

In Spark Streaming, existing work for performance optimization mainly includes the following aspects: adaptive batch size selection [24], [25]; online performance tuning [26]; and dynamic resource allocation [27]. However, there are few studies on the data skew. In particular, Liu et al. [16] presented a stream data prediction partitioner called SP-Partitioner. It uses the arrived batches of data as the prediction of key distribution of next batches of data. Whereafter, according to the prediction, SP-Partitioner generates a reference table to guide the allocation of next batches of data evenly. Compared with our work, SP-Partitioner relieves the reduce skew by improving the hash partition, and as a new breakthrough in this study, we focus on the data skew mitigation based on the range partition to balance the load of reduce tasks.

3. System Overview

In the design of Spark Streaming, the batches of stream are treated as traditional workloads, which are handled by the Spark engine to return the results in a near real-time way [28]. To address the reduce-side data skew for stream jobs, this paper optimizes the data assignment based on the range partition. Because it is suitable for all types of jobs, whereas the hash partition and other strategies can be applied to most jobs except those that require sorting [22]. The architecture of our system is shown in Fig. 2, which contains the following three steps:

Intermediate data prediction. In the proposed architecture, we use the outputted intermediate data of previous jobs to predicate that of the forthcoming job. During the shuffle phase of a batch processing, the frequency of each key of the intermediate data is counted and tracked, then we use the prediction model EWMA to forecast the size of key clusters.

Partition method optimization. To relieve the load imbalance of reduce tasks, a more reasonable data partition scheme is generated by ImRP. It involves solving the following issues: (1) how to determine the partition borders optimally; (2) how to split the border key cluster correctly when the semantics allow; and (3) how to allocate the work properly when the nodes are heterogeneous.

Intermediate data partition. When partitioning the intermediate data of the forthcoming job really, the system uses the data partition scheme obtained from the second step. In the shuffle phase, each key/value tuple in the map output gets its partition ID and then is written to relevant buffer array called buckets. In a map output, the key/value tuples with partition ID j compose the j^{th} bucket, which will become part of the j^{th} partition and be processed by the j^{th} reducer.

We will describe the realization of each part in detail in the following.

Table 1
Variable Declaration

$m, m \geq 1$	the number of map tasks;
$n, n \geq 1$	the number of reduce tasks;
$u, u \geq 1$	the number of key clusters of the intermediate data.
$< k_r, v_r >, r \geq 1$	a key/value tuple with key k_r of the intermediate data;
$b_{i,j}, 1 \leq i \leq m$	the j^{th} bucket whose data come from the i^{th} map task;
$(K_l, C_l), 1 \leq l \leq u$	a key cluster K_l whose number of key/value tuples is C_l;

4. Intermediate Data Prediction

To ascertain the distribution of intermediate data is an inevitable course to develop the balanced partition strategy [29]. For this purpose, most partition methods either pre-run the sampling of input data [6], [9], [14] or extract the intermediate data of partial map outputs for analysis [21], [30]. However, this may bring extra overhead. Making use of the data characteristics, this paper intends to forecast the key distribution for the forthcoming job. For illustrative purposes, some significant variables are declared in Table 1.

First of all, ImRP detects the key distribution of the intermediate data of previous jobs. Without loss of generality, we denote the forthcoming job as J_t that processes batch t, and the previous processed jobs as $\{J_{t-1}, J_{t-2}, \ldots, J_1 | t > 1\}$. There are some specific data structures can be formalized as follows:

(1) *BM*. A $m \times n$ matrix that denotes the outputs of all map tasks of a previous job, where the element $b_{i,j}$ indicates the j^{th} bucket whose data come from the i^{th} map task. Specifically, which bucket a key/value tuple belongs to is determined by the function *getPartitionID* according to the key, which can be formalized as:

$$b_{i,j} = \biguplus_{k_r \in KS_i : getPartitionID(k_r) = j} < k_r, v_r >, 1 \leq i \leq m, \tag{1}$$

where $< k_r, v_r >$ represents a key/value tuple, and KS_i is the key space of map output i.

(2) *DM*. A $(t-1) \times u$ matrix that represents the key distribution of the intermediate data of a series of previous jobs, which can be formalized as:

$$DM = \begin{bmatrix} (K_1^{t-1}, C_1^{t-1}) & (K_2^{t-1}, C_2^{t-1}) & \cdots & (K_u^{t-1}, C_u^{t-1}) \\ (K_1^{t-2}, C_1^{t-2}) & (K_2^{t-2}, C_2^{t-2}) & \cdots & (K_u^{t-2}, C_u^{t-2}) \\ \vdots & \vdots & \ddots & \vdots \\ (K_1^{1}, C_1^{1}) & (K_2^{1}, C_2^{1}) & \cdots & (K_u^{1}, C_u^{1}) \end{bmatrix},$$

where K_l represents a key cluster, and C_l is the number of tuples in the cluster of K_l. In particular, for the key distribution of J_{t-1}: $D_{t-1} = \{(K_1^{t-1}, C_1^{t-1}), (K_2^{t-1}, C_2^{t-1}), \ldots, (K_u^{t-1}, C_u^{t-1})\}$, ImRP designs a counter in each map output $[b_{i,1}, b_{i,2}, \ldots, b_{i,n}]$ to record the frequency of each key of the key/value tuples. After that, the key statistics of counters are combined to obtain the general key distribution.

There are many prediction algorithms in the literature, such as EWMA (Exponentially Weighted Moving Average) and CU-SUM [31]. Compared to the CUSUM model, the EWMA model gives different weights to historical and current values, which helps to better capture the changing trends. Moreover, because $\{(K_l^{t-1}, C_l^{t-1}), (K_l^{t-2}, C_l^{t-2}), \ldots, (K_l^1, C_l^1)\}$ are the typical time series data, it is more suitable to adopt the time series data prediction algorithm EWMA. Therefor, in the prediction of key cluster size for J_t, the frequency of a given key can be calculated as follows:

$$\begin{aligned} D_t^{\star}.C_l &= \alpha \times D_{t-1}.C_l^{t-1} + (1-\alpha) \times D_{t-1}^{\star}.C_l \\ &= \alpha \times \left(D_{t-1}.C_l^{t-1} + (1-\alpha) \times D_{t-2}.C_l^{t-2} + \ldots \right. \\ &\left. +(1-\alpha)^{t-2} \times D_1.C_l^1 \right), \end{aligned} \tag{2}$$

where K_l is shared by the key distribution D_{t-1} and the estimated key distribution D_{t-1}^{*}, $\alpha \in [0, 1]$ reflects a tradeoff between stability and

Table 2
rms error of key frequency estimation for each batch

	batch 2	batch 3	batch 4	batch 5	batch 6	batch 7	batch 8	batch 9	batch 10	average
EWMA	375.4	315.5	182.4	214.2	568.1	364.9	189.2	247.8	296.3	306.0
SP-Partitioner	214.7	424.1	244.9	517.5	964.8	457.1	253.4	232.9	483.6	421.4

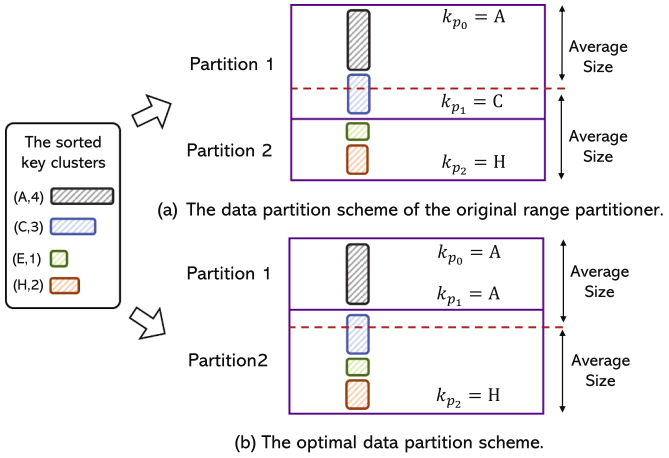


Fig. 3. Two different data partition schemes.

responsiveness. In our implementation, we set α to be 0.4 according to the evaluation result of section 7. In particular, if there is a key that is not shared by D_{t-1} and $D^{\star}_{t-1}$, the number of the un-contained part is set to 0. When dealing with the initial batch (i.e., $t = 1$), ImRP uses the original range partitioner for J_1.

We conduct a group of experiments to show how effective the EWMA model is. Considering the example of Fig. 1, we estimate the frequencies of 3462 keys of the batches by using the EWMA and the prediction model of SP-Partitioner [16] (directly use the previous batch as the prediction of the forthcoming batch). Table 2 is the statistics of the root mean square (*rms*) error of the key estimation for each batch, where $rms = \sqrt{\frac{\sum_{i=1}^{n} \Delta_i^2}{n}}$, n is the number of distinct keys, and Δ is the error between the estimated and the true value. The results show that EWMA can achieve higher accuracy than SP-Partitioner.

5. Partition Method Optimization

In this section, we generate the exact partition scheme based on the key estimation. For the purpose of avoiding data skew, a series of optimization measures are proposed, which involves the calculation of partition borders, split of border key clusters, and consideration of heterogeneous environments.

5.1. Calculation of Partition Borders

The data partition scheme of the range partitioner can be represented as a set of partition borders: $P = \{K_{p_0}, K_{p_1}, \cdots, K_{p_j}, \cdots, K_{p_n}\}$, where $K_{p_{j-1}} < K_{p_j}$, and K_{p_j} is a border key which represents the upper bound of the j^{th} partition. When the map tasks divide the intermediate data, the range of a partition $(K_{p_{j-1}}, K_{p_j}]$ indicates that the key/value tuples whose keys belonging to this range will be assigned to partition j. In particular, the 1^{st} partition includes key K_{p_0}. Assume that there are v key clusters dispatched in partition j, represented as: $\{(K_1, C_1), (K_2, C_2), \ldots, (K_v, C_v)\}$, the data size of partition j can be calculated as:

$$S_j = \sum_{l=1}^{v} C_l, K_{p_{j-1}} < K_l \le K_p. \tag{3}$$

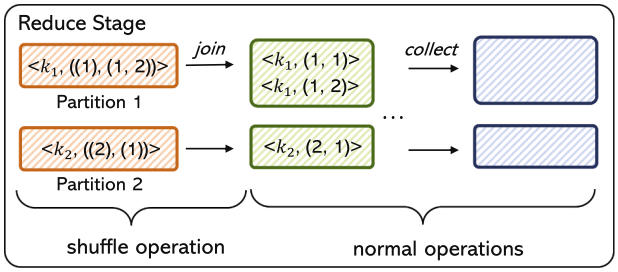


Fig. 4. Partition sizes before and after join operation.

In this method, our goal is to come up the partition borders appropriately that can make the partitions equaling in size.

It is obvious that the location of border keys has a direct impact on the balance of partitions. Here the original range partitioner takes a greedy approach: firstly, it sorts the key clusters of $D^{\star}_t$ in ascending lexicographic order according to their keys, and then calculates the average size of each partition. Thirdly, it puts the clusters of $D^{\star}_t$ sequentially into the partitions. In this process, if a cluster reaches or exceeds the average size of the partition, the key of the cluster is set to the partition border, and the next partition will be turned. However, the partition borders generated in this way is simple but not optimal.

Fig. 3 shows a simple example that there is a key distribution $D^{\star}_t$, in which the sorted key clusters are $(A, 4)$, $(C, 3)$, $(E, 1)$, and $(H, 2)$. The number of partitions is set to 2, so the average partition size is $(4+3+1+2)/2 = 5$. In accordance with the cutting method of the original partitioner, cluster A and C will be assigned to partition 1, and cluster E and H will be assigned to partition 2. The partition sizes are 7 and 3 respectively, as shown by Fig. 3 (a). However, we can easily find an optimal scheme that assigns cluster A to partition 1, and C, E and H to partition 2. The partition sizes are 4 and 6 respectively, as shown by Fig. 3 (b). Clearly, the latter partition scheme is more uniform than the former.

Moreover, the above schemes try to balance the partitions just for the shuffle operation, which may not guarantee the load balancing of the reducers throughout their execution. In the reduce stage, the execution of a reduce task includes both the shuffle operation and the normal operations [15]. As shown in Fig. 4, after the left-outer join operation, the number of key/value tuples changes and the partitions become unbalanced, even if the partition sizes are previously equal. Hence, it is necessary to make a trade-off between the balance of shuffle operation and that of normal operations.

To address these problems above, firstly, based on the points above, we formulate the integrated data size (*IS*) of the partition as follows:

$$IS_j = S_j + \lambda \times H(S_j), 1 \le j \le n, \tag{4}$$

where S_j is the size of partition j for the shuffle operation, $H(S_j)$ denotes the partition size for the normal operations, and λ is the ratio of normal operations workload to shuffle operation workload. There are many factors that can affect workload, such as the tuple numbers and the complexity of computation. Measuring the workload in a comprehensive way will make the partition more balanced. For the sake of generality, we set $\lambda = 1$, which means that we measure the workload solely based on the number of key/value tuples. Further, $H(S_j)$ can be estimated as:

$$H(S_j) = ratio_j \times S_j, 1 \le j \le n, \tag{5}$$

Require:
The sorted key cluster distribution: $D_t^{\star} = \{(K_1, C_1), (K_2, C_2), \cdots, (K_u, C_u)\}$;
The number of partitions: n.
Ensure :
The border keys: $P = \{K_{p_0}, K_{p_1}, \cdots, K_{p_n}\}$.

```
initialize the first and last border keys: K_{p_0} = K_1; K_{p_n} = K_u;
int minSize = 0, maxSize = 0;
for l = 1; l ≤ u; l++ do
    maxSize += C_l + H(C_l);
end
while minSize <= maxSize do
    midSize = (minSize + maxSize)/2;
    if check(midSize, D_t^*, n) then
        maxSize = midSize − 1;
    end
    else
        minSize = midSize + 1;
    end
end
int curSize = 0, j = 1;
for l = 1; l ≤ u; l++ do
    if curSize + C_l + H(C_l) < minSize then
        curSize += C_l + H(C_l);
    end
    else if curSize + C_l + H(C_l) == minSize then
        K_{p_j} = K_l; curSize = 0; j++;
    end
    else
        K_{p_j} = K_{l−1}; curSize = C_l + H(C_l); j++;
    end
end
return P.
```

Algorithm 1. Calculation of Partition Borders

Require: The intermediate value of the size: *midSize*;
The sorted key cluster distribution: $D_t^{\star}$;
The number of partitions: *n*.
Ensure : A boolean value.

```
1  int l = 1, num = 1, curSize = 0;
2  while l <= u do
3      if curSize + C_l + H(C_l) <= midSize then
4          curSize+ = C_l + H(C_l); l + +;
5      end
6      else if C_l + H(C_l) <= midSize then
7          num + +; curSize = C_l + H(C_l); l + +;
8      end
9      else
10         return false;
11     end
12 end
13 return num <= n.
```

Algorithm 2. The check function

where $ratio_j$ denotes the ratio of the partition size after and before the shuffle operation, which can be calculated by the partitions of J_{t-1}. Hence, $H(S_j)$ can be calculated as:

$$\begin{aligned} H(S_j) &= H\left(\sum_{l=1}^{v} C_r\right) \\ &= ratio_j \times \sum_{l=1}^{v} C_r = \sum_{l=1}^{v} ratio_j \times C_r. \end{aligned} \tag{6}$$

On the other hand, $H(\sum_{l=1}^{v} C_r) = \sum_{l=1}^{v} H(C_r)$. Here $H(C_l)$ represents the size of the key cluster after the shuffle operation, which can be precisely defined by some specific shuffle operators. For example, for the widely used self join operator, it can be estimated as: $H(C_l) = (C_l)^2$. And for the reduceBykey operator, it can be defined as: $H(C_l) = 1$. Whereas for the sort operator commonly seen in database applications, the function can be defined as: $H(C_l) = C_l$. Because it does not change the amount of data of partitions.

Secondly, we propose a novel method to calculate the partition borders optimally. The integrated size of partition IS_j is used in this model. In the first place, because the execution time of parallel tasks is usually decided by the most heavily loaded task, the partition balance problem can be transformed as minimizing the maximum data size among all the partitions. As a result, the objective function of this model can be specified as:

$$\begin{aligned} &\min\left\{\max_{j=1,2,\cdots,n} IS_j\right\} \\ &= \min\left\{\max_{j=1,2,\cdots,n}\left(\sum_{l=p_{j-1}+1}^{p_j}\left(C_l + H(C_l)\right)\right)\right\}. \end{aligned} \tag{7}$$

This paper uses the dichotomy method to resolve the *Maximum-Minimum* problem above. Algorithm 1 describes the process of calculating these partition borders. To elaborate the algorithm in detail, some significant variables are explained as follows:

(1) *maxSize*. The maximum integrated data size of the partitions. At this point, we put all the key clusters of $D_t^{\star}$ into one partition.

(2) *minSize*. The minimum integrated data size of the partitions. This means that we would not put any clusters into one of the partitions, so the value of *minSize* is 0.

(3) *midSize*. The median of *maxSize* and *minSize*. The purpose of setting this variable is to search for a suitable data size, which represents the maximum amount of data each partition allowed to carry. Since if this threshold is large, it does not need *n* partitions to accommodate all the key clusters; and if this threshold is small, the number of partitions required will be greater than *n*.

As shown in Algorithm 1, firstly, the algorithm determines the value of *maxSize* and *minSize*, as lines 2-5; Secondly, it uses the dichotomy to take the median of the two sizes (i.e., *midSize*) and uses the *check* function to determine whether this value is adjusted up or down. The details of the *check* function is shown by Algorithm 2. This process is terminated until *minSize* equals *maxSize*, and then we obtain the suitable size *minSize*, as lines 6-14; Thirdly, the algorithm uses *minSize* as the threshold to divide the clusters into *n* partitions and calculate the partition borders. In particular, it puts the clusters of $D_t^{\star}$ sequentially into partitions as much as possible, and ensuring that the size of each partition does not exceed this threshold, as lines 15-26; Finally, return the partition borders *P*. The average time complexity of Algorithm 1 is $O(u\log Sum(pi))$, where *u* is the number of clusters of $D_t^{\star}$, and *pi* represents the probability of each situation.

5.2. Split of Border Key Clusters

The range partitioner of Spark assigns a key cluster only to a partition, but this may yet lead to even the best data partition scheme cannot make the partitions absolutely equal [22]. As the example of section 5.1,

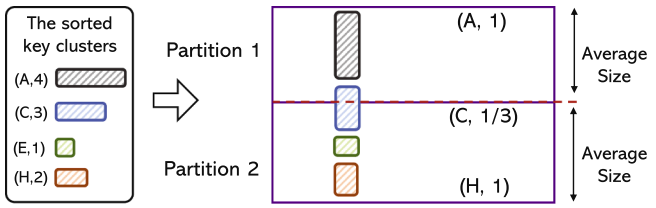


Fig. 5. Split of border key cluster in data partition scheme.

with this constraint, the sizes of two partitions are still unequal under the optimal data partition scheme. Nevertheless, for some shuffle operators, such as sort and union (the union of two sets) that treat each intermediate key/value tuple independently in the reduce stage, assigning the same keys to multiple partitions does not affect the final result. Hence, in view of the great flexibility, we provide a mechanism of splitting border key cluster in the range partition of Spark.

In ImRP, the partition border additionally comes with a proportion value to split its cluster, i.e., border key K_{p_j} becomes tuple (K_{p_j}, f_{p_j}), $0 < f_{p_j} \leq 1$, where f_{p_j} is the proportion of key cluster K_{p_j} divided into partition j. For the example described in section 5.1, Fig. 5 shows the usage of the cluster split to divide the two partitions evenly. The data partition scheme is $P = \{(A, 1), (C, 1/3), (H, 1)\}$, so the size of both two partitions is 5. Consequently, when partitioning the intermediate data really, the key/value tuple whose key is C will be assigned to partition j with the probability of 1/3, and to the next partition with the probability of 2/3. By this means, a large number of related tuples will be distributed proportionally to the partitions, and the skewness can be eliminated thoroughly.

It is worth noting that the split of key cluster can only be applicable to these shuffle operators whose semantics allow, which can be decided by setting a parameterized flag when the job begins execution.

Algorithm 3 shows how the partition borders are calculated with cluster split, and we consider the integrated partition size. Firstly, the algorithm calculates the total integrated data size and the average integrated size of the partitions (i.e., *avgSize*), as lines 3-6; Then it puts the key clusters of $D_t^\star$ into the partitions in order (starting with partition 1) and records the size for the current partition (i.e., *curSize*). In this process, if the current cluster is placed so that *curSize* is smaller than *curSize*, continue to place the next cluster, as lines 7-11; Meanwhile, if the current cluster is put so that *curSize* just reaches *curSize*, then the key of the current cluster is set as the border key, and the proportion value is set to 1, as lines 12-14; Otherwise, set the key of the current cluster as the border key, and calculate the corresponding proportion value, as lines 15-19; For the latter two cases, the next partition will be turned. Finally, return the partition borders P, as line 20. The time complexity of algorithm 3 is $O(u)$, where u is the number of the key clusters in $D_t^\star$.

Require: The sorted key cluster distribution: $D_t^\star = \{(K_1, C_1), (K_2, C_2), \cdots, (K_u, C_u)\}$;
The number of partitions: n.
Ensure : The border keys: $P = \{(K_{p_0}, f_{p_0}), (K_{p_1}, f_{p_1}), \cdots, (K_{p_n}, f_{p_n})\}$.

```
1  initialize the first and last border keys: K_{p_0} = K_1, f_{p_0} = 1; K_{p_n} = K_u, f_{p_n} = 1;
2  initialize the total integrated size and average size: totalSize = 0; avgSize = 0;
3  for l = 1; l ≤ u; l++ do
4      totalSize += C_l + H(C_l);
5  end
6  avgSize = totalSize/n;
7  int curSize = 0; j = 1;
8  for l = 1; l ≤ u; l++ do
9      if curSize + C_l + H(C_l) < avgSize then
10         curSize += curSize + C_l + H(C_l);
11     end
12     else if curSize + C_l + H(C_l) == avgSize then
13         K_{p_j} = K_l; f_{p_j} = 1; curSize = 0; j++;
14     end
15     else
16         K_{p_j} = K_l; f_{p_j} = (avgSize − curSize)/(C_l + H(C_l));
17         curSize = C_l + H(C_l) − (avgSize − curSize); j++;
18     end
19 end
20 return P.
```

Algorithm 3. Calculation of Partition Borders with Cluster Split

5.3. Consideration of Heterogeneous Environments

The data partition scheme generated so far, based on the assumption of homogeneous environments, aims to make the partitions similar in size, thus the reduce tasks with the same load can be completed at the same time. While in a heterogeneous environment, the capabilities of nodes can be different for various reasons, such as heterogeneous resource capacity, resource competition, network congestion, etc [32]. Therefore, in such an environment, it is necessary to consider the node's performance in data assignment instead of always dividing them evenly.

The basic idea of ImRP is to allocate the quantity of data to reduce tasks according to the capability of each node. It requires an indicator to measure the performance of nodes appropriately. Previous researches focusing on this point are mostly designed for Hadoop [30], [33], which may be inaccurate for Spark tasks. In this paper, we use the average processing speed of reduce tasks running on a node, which can be formulated as:

Algorithm 4. Get Partition ID

Require: The data partition scheme: $P = \{(K_{p_0}, f_{p_0}), (K_{p_1}, f_{p_1}), \cdots, (K_{p_n}, f_{p_n})\}$;
The key/value tuple to be partitioned: (k_r, v_r).
Ensure : The partition ID for tuple (k_r, v_r): q.

```
1  //random() returns a uniform [0,1] random number;
2  for int j = 1; j ≤ n; j++ do
3      if k_r < K_{p_j} then
4          q = j; break;
5      end
6      else if k_r == K_{p_j} then
7          if f_{p_j} == 1 then
8              q = j; break;
9          end
10         else
11             d = random();
12             if d ≤ f_{p_j} then
13                 q = j; break;
14             end
15             else
16                 q = j + 1; break;
17             end
18         end
19     end
20 end
21 return q.
```

$$capability_j = \frac{data\_size_j}{execution\_time_j}, \tag{8}$$

where $data\_size_j$ denotes the total data size processed by the reduce tasks, and $execution\_time_j$ denotes the total execution time of these reduce tasks running on node j. In Spark Streaming, the capability of the node can be calculated by the finished tasks of previous jobs.

Then we formulate the load balancing problem in heterogeneous environments as follows: firstly, the relative capability of the node can be defined as:

$$relative\_capability_j = \frac{capability_i}{avg\_capability}, \tag{9}$$

where $avg\_capability$ denotes the average capability of all the worker nodes. Then, assume that partition j (processed by reduce task j) to be handled on worker node j. As a result, the objective function of the model is modified as:

$$\begin{aligned} &\min\left\{\max_{j=1,2,\cdots,n}\frac{IS_j}{relative\_capability_j}\right\} \\ &= \min\left\{\max_{j=1,2,\cdots,n}\left(\frac{\sum_{l=p_{j-1}+1}^{p_j}\left(C_l + H(C_l)\right)}{relative\_capability_j}\right)\right\}. \end{aligned} \tag{10}$$

For this *Maximum-Minimum* problem, we can use the same algorithm described in section 5.1 to calculate the optimal data partition scheme in the heterogeneous environment. Moreover, if the split of border key cluster is allowed, the load of reduce tasks can be enhanced further.

6. Intermediate Data Partition

When partitioning the intermediate data of job t in the shuffle phase, Algorithm 4 describes how the key/value tuples get their partition ID according to the generated data partition scheme. For generality, the partition borders without key cluster split can be regarded as a special case of the partition borders with key cluster split, i.e., border key K_{p_j} can be seen as border key $(K_{p_j}, 1)$. The algorithm uses the sequential search method to get the partition ID. For a specific key/vaule tuple (k_r, v_r), it starts looking at partition border K_{p_1} until finds a partition border that is larger than or equal to k_r. Then, if k_r less than K_{p_j}, the partition ID is set to j, as lines 2-5; Otherwise (i.e., k_r equals K_{p_j}), there are two cases: first if f_{p_j} is 1, set the partition ID as j, as lines 6-9; otherwise it uses a random number and the value of f_{p_j} to determine whether the tuple is assigned to the current partition j or the next partition $j+1$, as lines 10-20; Finally, return the partition ID, as line 21. The time complexity of algorithm 4 is $O(n)$, where n is the number of partitions.

7. Evaluation

In this section, we evaluate the performance of the proposed partitioner. We have implemented ImRP in the source codes of Spark-core 3.0 project. Hence, the task progress can use our achievement by invoking the *getPartitionID* method provided by ImPR. The execution of the shuffle phase still depends on Spark's original mechanism.

7.1. Experiment Setting

Our test cluster consists of 9 physical machines. Each machine is equipped with Intel Xeon processor E7-8867 v4, 2.4GHz, 64GB RAM and 512GB of disk. These physical machines are organized in three racks connected by 1Gbps Ethernet and managed by OpenStack cloud operating system [34]. We use the KVM virtualization software [35] to

Table 3
Benchmarks and Application Types

Benchmarks	Application types
WordCount	Simple job
Sort	Simple job
PageRank	Search engine application
LDA	Machine learning application

construct medium sized VMs with 4 virtual cores, 8GB RAM, and 64 GB of disk space. First of all, we conduct our experiments in a homogeneous environment where each machine runs two virtual machines, and then we create a heterogeneous environment for testing.

To estimate the performance, as shown in Table 3, two representative micro-benchmarks: *WordCount* and *Sort*, and two representative macro-benchmarks: *PageRank* and *LDA* (Latent Dirichlet Allocation) are selected. The application of micro-benchmarks contains one job with two stages (i.e., map stage and reduce stage), so that we can easily observe the effect of data partition on the execution time of reduce stage. By contrary, the application of macro-benchmarks contains one or more jobs with multiple stages.

In our experiments, the following data partition strategies are chosen for comparison:

OrigRange: (The Original Range partitioner [36]). As one of the default partition methods, *OrigRange* is suitable for all types of jobs and can alleviate the reduce skew to some extent compared to the Hash partitioner. It runs an extra job ahead of time to estimate the key distribution, but this occurs extra time overhead.

SASM: (Spark Adaptive Skew Mitigation [37]). From another point of view, *SASM* mitigates skew of shuffle read and computation misdistribution dynamically with metadata collected beforehand. When a new task is registered, it re-partitions the unprocessed blocks of straggling tasks to other idle tasks.

SP-Partitioner: (A stream data prediction partitioner [16]). It treats the arrived batches as candidate samples and uses it to predict the characteristics of intermediate data. According to this, *SP-Partitioner* generates a reference table to guide the allocation of next batches of data evenly. It is an improved hash partition.

The following indicators are used for performance evaluation:

• *Job execution time*: The time from the start to the end of a stream job. Since there are some pre-work in the above strategies and the extra overhead cannot be ignored, this indicator can reflect the overall performance of different methods fairly.

• *Reduce stage execution time*: It refers to the time when the reduce tasks start to fetch intermediate data in the shuffle phase to the end of the task. In Spark, the tasks are executed stage by stage. This means that the tasks of the child stage will not be started until the parent stage complete execution.

• *Coefficient of variation (CV)*: It is a common measurement for the data skew: $CV = \frac{stddev(\vec{x})}{mean(\vec{x})}$, where $\vec{x}$ is a vector that contains the integrated data size processed by each task [22]. Larger coefficient indicates heavier skew.

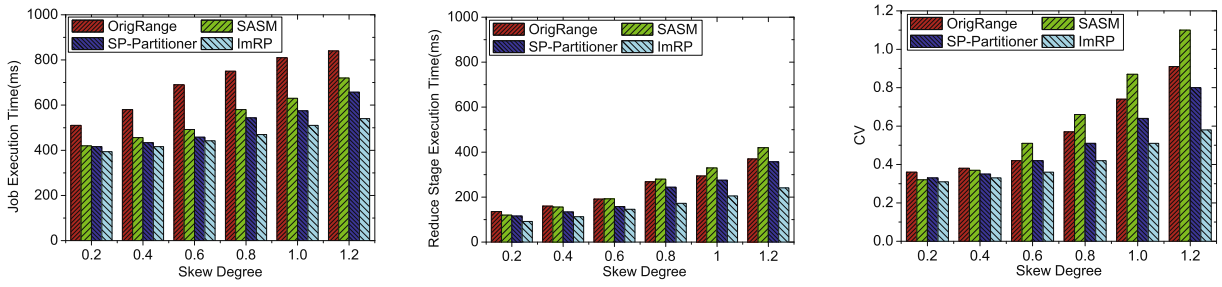
To reduce the interference of variable environments, we take the average of 60 consecutive stream jobs for the above indicators in Spark Streaming.

7.2. Performance

7.2.1. Micro-Benchmarks

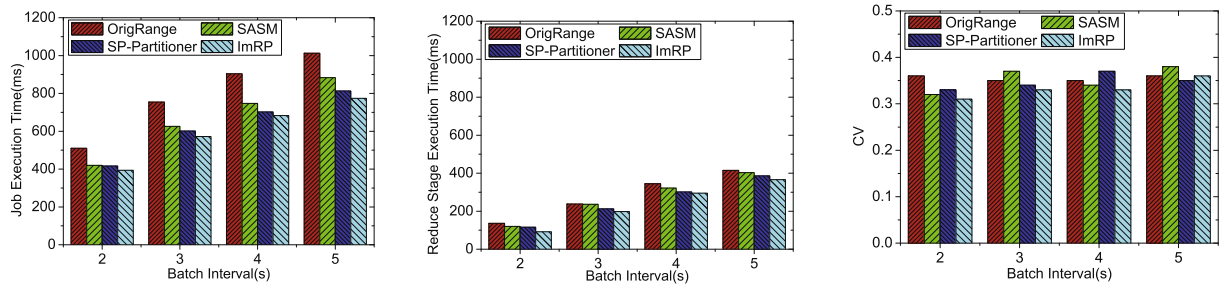
WordCount is widely used in Mapreduce system processing, which counts the number of occurrences of each word in input data. We use the *groupByKey* operator in the program. To test the performance under different distribution of input batch, the synthetic data stream is used and generated by *RandomWriter* whose data follows the *Zipf* distribution. It can represent many real-world data distribution [38]. The *Zipf* distribution uses the parameter σ to control the skew degree: a larger σ indicates a more uneven distribution of input batch. The data arrival speed is set to 80MB/s and the batch interval is set to 2s for the stream jobs.

Fig. 6 shows the performance comparison of different methods (*OrigRange, SASM, SP-Partitioner* and *ImRP*) when σ varies from 0.2 to 1.2. As shown in Fig. 6(a), the job execution time of all methods



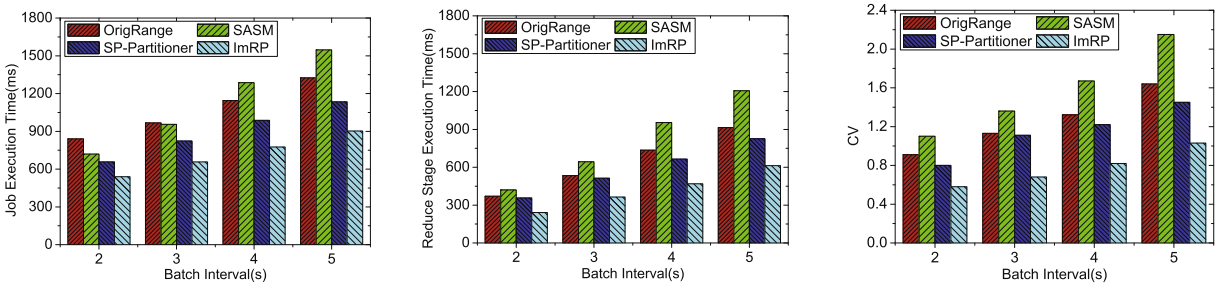
(a) Job execution time. (b) Reduce stage execution time. (c) Coefficient of variation.

Fig. 6. Performance with different skew degrees of input batch under wordCount.



(a) Job execution time. (b) Reduce stage execution time. (c) Coefficient of variation.

Fig. 7. Performance with different batch intervals under wordCount ($\sigma = 0.2$).



(a) Job execution time. (b) Reduce stage execution time. (c) Coefficient of variation.

Fig. 8. Performance with different batch intervals under wordCount ($\sigma = 1.2$).

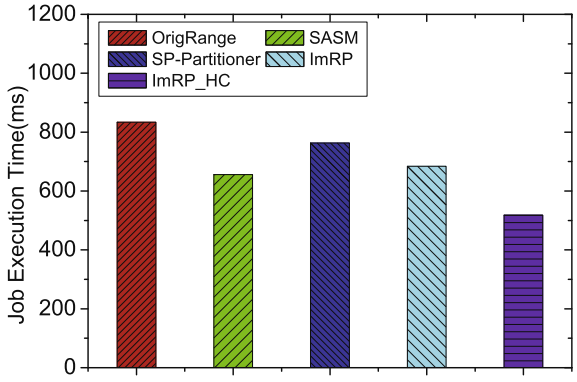


Fig. 9. Performance with different skew degrees of input batch under sort.

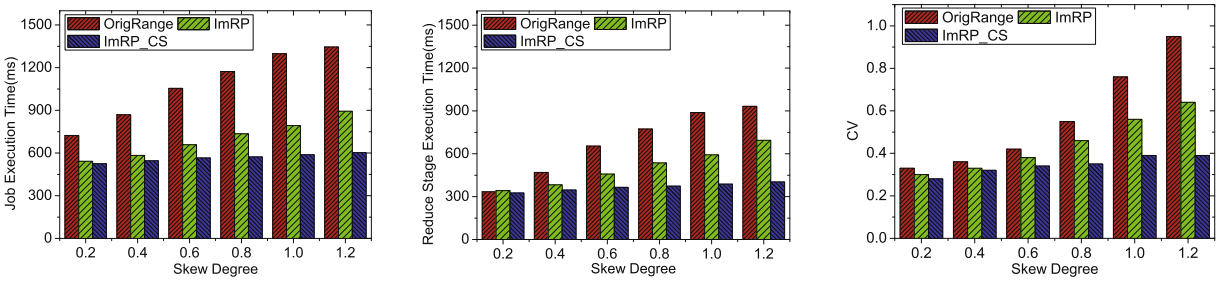
increases with the increase of σ, and *ImRP* can obtain better performance than other methods. In particular, when σ is 1.2, *ImRP* can decrease the job execution time by 35.7%, 25%, and 17.8% compared to *OrigRange, SASM*, and *SP-Partitioner*, respectively. In this case, the maximum job execution time of *OrigRange, SASM, SP-Partitioner*, and *ImRP* is 1238ms, 957ms, 784ms, and 646ms respectively, and the minimum job execution time for these methods is 683ms, 567ms, 486ms, and 425ms respectively. Further, Fig. 6(b) illustrates that the job performance gap is mainly caused by the difference in the duration of reduce stage. It is worth noting that the stage time of *OrigRange* is generally less than that of *SASM*, but it takes more time to run the job because of the delay of pre-run job. This also demonstrates the negligible extra overhead of *ImRP*. Fig. 6(c) shows the coefficient of variation. It verifies that *ImRP* can achieve good partition balance for reduce tasks through its data partition scheme, which is the main factor in improving the performance.

In order to estimate the performance of relevant methods under different data volume, we also set up different batch intervals for the stream jobs in the cases of slight ($\sigma = 0.2$) and severe ($\sigma = 1.2$) skewed input batch. Fig. 7 depicts the experimental results when the batch interval varies from 2s to 5s and $\sigma = 0.2$. Specifically, Fig. 7(a) and Fig. 7 (b) show that the job and reduce stage execution time of all methods increase gradually as the data size increases, but *ImRP* performs best on these two indicators. In particular, when the batch interval is 5s, *ImRP* can reduce the reduce stage execution time by 11.8%, 9.1%, and 5.2% compared to *OrigRange, SASM*, and *SP-Partitioner*, respectively. It observes that all the methods can balance the partitions well, as shown by Fig. 7(c).

Fig. 8 shows the performance with different batch intervals under heavily skewed input batch. The bars in Fig. 8(a) explain that *ImRP* and *SP-Partitioner* can get lower job and reduce stage execution time than *OrigRange* and *SASM*. However, *ImRP* still outperform *SP-Partitioner*. In addition, when the batch interval equals or exceeds 4s, the job execution time of *SASM* is greater than that of *OrigRange*. This reason can be explained as *SASM* only re-assigns unprocessed data of slow tasks, therefore, it cannot re-arrange the intermediate data when more and more data is produced and handled. Fig. 8(c) shows that *ImRP* can partition the data more evenly than others even under the serious skewed input batch. Particularly, when the batch interval is 5s, the *CV* value of *OrigRange, SASM, SP-Partitioner*, and *ImRP* is 1.64, 2.15, 1.45, and 1.03 respectively, the maximum *CV* value of these methods is 2.13, 3.75, 1.88, and 1.46 respectively, and the minimum *CV* value of these methods is 1.47, 1.89, 0.85 and 0.73 respectively.

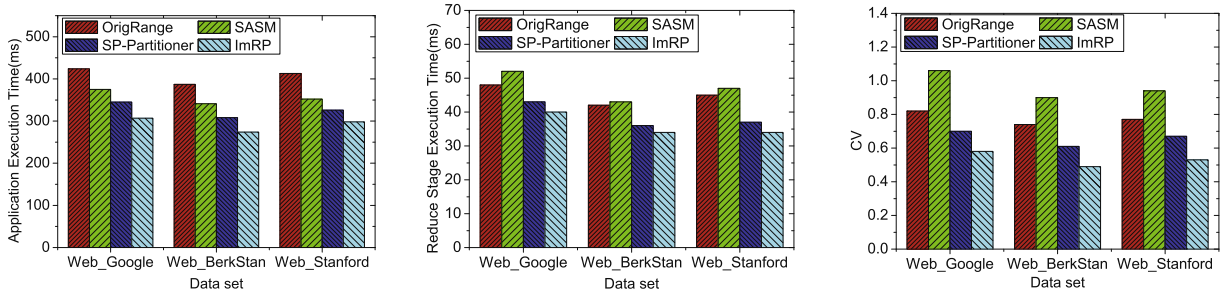
Sort is common in the Spark workload testing that makes the data objects orderly. In the Spark engine, only the range partition can implement this function. Since *SASM* does not consider this type of jobs and *SP-Partitioner* is based on the hash partitioner, they are not suitable for the sort job. Moreover, to assess how the key cluster split can benefit the performance, we set up a group of comparative experiments: *ImRP* without the border key cluster split and with the border key cluster split (marked as *ImRP_CS*). Like wordCount, the experiments run the synthetic data stream, and the *sortByKey* shuffle operator is utilized in the program. We set the data arrival speed to 60MB/s and the batch interval to 2s for the stream jobs.

Fig. 9 (a) and Fig. 9(b) show that *ImRP* and *ImRP_CS* can perform much better than *OrigRange* in terms of the job and reduce stage execution time. Moreover, *ImRP_CS* with the border key cluster split can



(a) Job execution time. (b) Reduce stage execution time. (c) Coefficient of variation.

Fig. 10. Performance with different data sets under pageRank.



(a) Application execution time. (b) Reduce stage execution time. (c) Coefficient of variation.

Fig. 11. Performance with different data sets under LDA.

Table 4
Heterogeneous Environment

Level	Hosts	VMs
2VMs/host	5	10
3VMs/host	3	9
4VMs/host	1	4
Total	9	23

improve the performance on the basis of *ImRP*: when $\sigma = 1.2$, *ImRP_CS* can shorten the reduce stage time by 41.9% over *ImRP*. The reason behind it is that *ImRP_CS* allowing key cluster split can ensure the partitions completely equal in the ideal case. As can be seen from Fig. 9(c), the *CV* value of *ImRP_CS* is essentially constant in all cases and remains at a low level.

7.2.2. Macro-Benchmarks

In these experiments, two real-world applications: pageRank and LDA (Latent Dirichlet Allocation) are selected from the examples of Spark for testing. We rewrite the two programs so that they can process the batches of stream and return the results in a real-time fashion. Because the applications contain one or more jobs with multiple stages, we use the application execution time in the evaluation.

PageRank is a well-known algorithm in search domain that ranks pages according to their importance. We run the experiments on three real data sets: *Web_Google, Web_BerkStan*, and *Web_Stanford* from [39], which are read as HDFS file streams. The data arrival speed is set to 1MB/s as it spends lots of time on iterative computation, and the batch interval is set to 2s for the stream jobs. We set the parameter *numIterations* = 10 in the program, so the application contains 1 job and 13 stages.

Fig. 10 shows us the experiment results of relevant methods. The rectangles of Fig. 10(a) illustrate that *ImRP* can improve the application performance effectively. In particular, on the *Web_Google* data set, *ImRP* can finish applications 27.6%, 18.1% and 11% faster than *OrigRange, SASM* and *SP-Partitioner*, respectively. Furthermore, to understand how *ImRP* balances the partitions across the reducers, we observe one of the

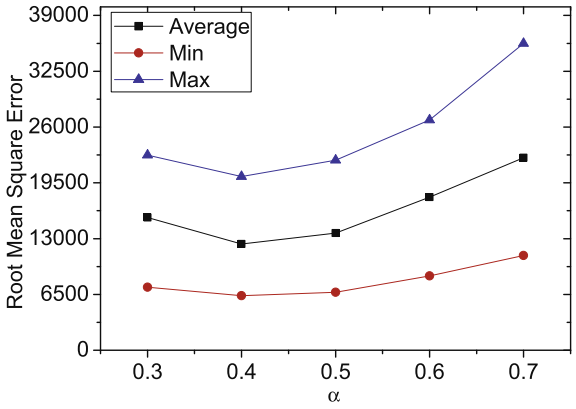
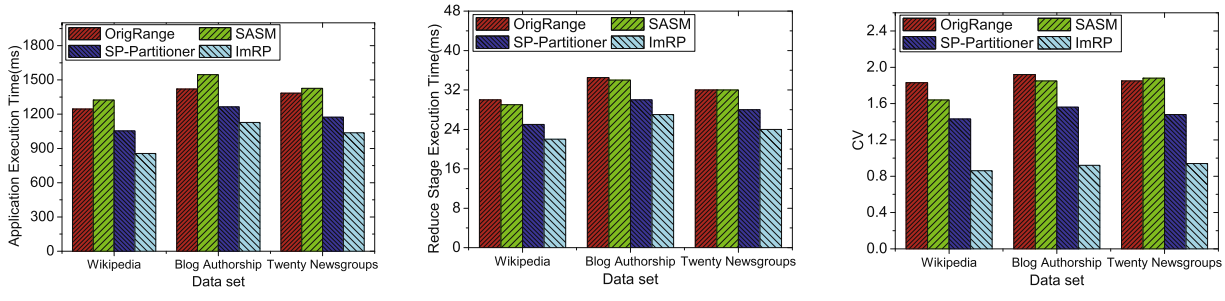


Fig. 13. *rms* error with different value of α.

reduce stages (i.e., the third stage of pageRank) that contains the *join* operation and plot these stage execution time and *CV* of different methods as Fig. 10(b) and Fig. 10(c), respectively. It further validates that by optimizing the data partition scheme, *ImRP* can enhance the execution efficiency of the stage than other methods.

LDA is an important algorithm in NLP (Natural Language Processing) for topic modelling. It extracts some keywords that can express each topic from a large number of documents. In these experiments, three kinds of real data sets are used: *Wikipedia Corpus* [7], *Blog Authorship Corpus* [40], and *Twenty Newsgroups* [41]. The program sets the number of topics to 5 and *numIterations* = 10, so the application has 26 jobs, a total of 90 stages. The data arrival speed is set to 1MB/s and the batch interval is set to 2s for the stream jobs.

Fig. 11 (a) shows the application execution time of the methods. Though some of the stages of LDA application are short, the cumulative performance gains of many stages can improve the overall performance obviously. As we can see, on the *Wikipedia Corpus* data set, *ImRP* can finish applications 31.2%, 35.3% and 18.7% faster than *OrigRange,*



(a) Application execution time. (b) Reduce stage execution time. (c) Coefficient of variation.

Fig. 12. Performance in heterogeneous environments.

SASM and *SP-Partitioner*, respectively. As before, we draw Fig. 11(b) and Fig. 11(c) to analyze how *ImRP* behaves at the stage (i.e., the third stage of the first job). The stage contains the *reduceByKey* operator that is often used in practice. Since *ImRP* estimates the frequency of keys after the map-side aggregation, it is more accurate than other algorithms. Hence, the better partition balance can be obtained by *ImRP*.

7.2.3. Heterogeneous Environments

In these experiments, we evaluate the performance of the proposed partitioner in heterogeneous environments. As shown in Table 4, the Spark cluster is created by running different numbers of virtual machines on the 9 physical machines. Under such an environment, the performance of the VMs can vary significantly due to the workloads from the co-hosted VMs [30].

We run the wordCount benchmark on the synthetic data stream. For comparison, two versions of *ImRP* are designed: *ImRP* without heterogeneity consideration and with heterogeneity consideration (marked as *ImRP_HC*). Fig. 12 shows the results for the five methods (*OrigRange, SASM, SP-Partitioner, ImRP* and *ImRP_HC*) when $\sigma = 0.8$. As we can see from the figure, *ImRP_HC* with the consideration can finish jobs 24.3% faster than *ImRP*. Moreover, it can finish jobs 37.9%, 21%, and 32.1% faster than *OrigRange, SASM*, and *SP-Partitioner*, respectively. The experiments indicate that *ImRP* fits well in the heterogeneous cluster.

7.3. Parameter Analysis

In section 4, we propose to utilize the EWMA model to predict the size of the key clusters. The EWMA has a parameter α that reflects a trade-off between the stability and responsiveness. In this section, we evaluate the accuracy of the key estimation with different α values to see the variance trends. The experiments run the wordCount benchmark on the synthetic data stream ($\sigma = 0.8$). Fig. 13 shows the average *rms* error, min *rms* error, and max *rms* error for the key frequency estimation of 10 jobs that process 10 batches. It can be seen that when α is set to 0.4, *ImRP* obtains the lowest average *rms* error. So we empirically set the value of α to 0.4. Besides, under this setting, the EWMA can be more accurate than *SP-Partitioner*, which was verified by Table 2.

8. Conclusions

In Spark Streaming computing environments, the default partition methods easily cause the load imbalance of reduce tasks in the intermediate data allocation. This situation extends the execution time of stages. This paper attempts to extenuate the data skew among the reducers for stream jobs. First, it predicts the key distribution for the forthcoming job according to the previous batch processing. Then it generates the appropriate partition strategy, in which a series of optimization measures are proposed, including calculating the partition borders optimally, splitting the border key cluster if necessary, and allocating the work properly when the nodes are heterogeneous.

The experiment results based on several representative ben-chmarks, which are prone to data skew, verify that by considering the partition balance before and after shuffle operation, the load imbalance of the reduce tasks can be addressed by the proposed partitioner successfully. Moreover, because the data partition scheme is produced in advance, it has no extra delay for the job execution.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

The work is supported by the National Key Research and Development Program of China (2017YFB02018YFB1701400, 2018YFB0203804), the National Natural Science Foundation of China (Grant Nos. 61873090, L1824034, L1924-056), Ministry of Education-China Mobile Research Fund Proj-ect(MCM20170506), China Knowledge Centre for Engineering Sciences and Technology Project(CKCEST-2018-1-13, CKCE-ST-2019-2-13).

References

[1] W. Wingerath, F. Gessert, S. Friedrich, N. Ritter, Real-time stream processing for big data, it - Information Technology 58 (4) (2016) 186–194.
[2] Spark streaming, https://spark.apache.org/streaming/.
[3] M. Zaharia, An architecture for fast and general data processing on large clusters, in: Ph.D. thesis, EECS Department, University of California, Berkeley (Feb 2014). http://www2.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-12.html.
[4] Apache storm, http://storm.apache.org/.
[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M.J. Franklin, J.M. Hellerstein, W. Hong, S. Krishnamurthy, S.R. Madden, F. Reiss, M.A. Shah, Telegraphcq: continuous dataflow processing. ACM SIGMOD International Conference on Management of Data, 2003.668–668
[6] Y. Xu, W. Qu, Z. Li, Z. Liu, C. Ji, Y. Li, H. Li, Balancing reducer workload for skewed data using sampling-based partitioning, Computers & Electrical Engineering 40 (2) (2014) 675–687.
[7] Wikipedia corpus, https://www.english-corpora.org/wiki/.
[8] J. Berlinska, M. Drozdowski, Comparing load-balancing algorithms for mapreduce under zipfian data skews, Parallel Computing 72 (2018) 14–28, https://doi.org/10.1016/j.parco.2017.12.003.
[9] Y. Xu, P. Zou, W. Qu, Z. Li, K. Li, X. Cui, Sampling-based partitioning in mapreduce for skewed data, 2012, Chinagrid Conference. 1–8.
[10] Y.C. Kwon, M. Balazinska, B. Howe, J. Rolia, Skewtune:mitigating skew in mapreduce applications, Proceedings of the Vldb Endowment 5 (12) (2012) 25–36.
[11] Y. Le, J. Liu, F. Ergun, D. Wang, Online load balancing for mapreduce with skewed data input. INFOCOM, 2014 Proceedings IEEE, 2014, pp. 2004–2012.
[12] L. Chen, W. Lu, X. Che, W. Xing, L. Wang, Y. Yang, Mrsim: Mitigating reducer skew in mapreduce. International Conference on Advanced Information NETWORKING and Applications Workshops, 2017, pp. 379–384.
[13] Apache hadoop, http://hadoop.apache.org/.
[14] Z. Tang, X. Zhang, K. Li, K. Li, An intermediate data placement algorithm for load balancing in spark computing environment, Future Generation Computer Systems 78 (2018) 287–301.Pt.1
[15] Z. Tang, W. Lv, K. Li, K. Li, An intermediate data partition algorithm for skew mitigation in spark computing environment, IEEE Transactions on Cloud Computing (2018), https://doi.org/10.1109/TCC.2018.2878838.1–1
[16] G. Liu, X. Zhu, W. Ji, D. Guo, W. Bao, G. Hui, Sp-partitioner: A novel partition method to handle intermediate data skew in spark streaming, Future Generation Computer Systems 86 (2018) 1054–1063.
[17] Y. Xu, P. Kostamaa, A new algorithm for small-large table outer joins in parallel DBMS. Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, 2010, pp. 1018–1024.March 1-6, 2010, Long Beach, California, USA
[18] S. Acharya, P.B. Gibbons, V. Poosala, Congressional samples for approximate answering of group-by queries. Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, 2000, pp. 487–498, https://doi.org/10.1145/342009.335450.Dallas, Texas, USA.
[19] A. Shatdal, J. Naughton, Adaptive parallel aggregation algorithms, 1995, 24, 104–114, 10.1145/568271.223801.
[20] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (2008) 107–113, https://doi.org/10.1145/1327452.1327492.
[21] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, S. Wu, Handling partitioning skew in mapreduce using leen, Peer-to-Peer Networking and Applications 6 (4) (2013) 409–424.
[22] Q. Chen, J. Yao, Z. Xiao, Libra: Lightweight data skew mitigation in mapreduce, IEEE Transactions on Parallel & Distributed Systems 26 (9) (2015) 2520–2533.
[23] B. Gufler, N. Augsten, A. Reiser, A. Kemper, Handling data skew in mapreduce. Closer 2011 - Proceedings of the International Conference on Cloud Computing and Services Science, Noordwijkerhout, Netherlands, 2012, pp. 574–583.7-9 May
[24] W. Li, D. Niu, Y. Liu, S. Liu, B. Li, Wide-area spark streaming: Automated routing and batch sizing, 2017.
[25] D. Cheng, Z. Xiaobo, W. Yu, J. ChangJun, Adaptive scheduling parallel jobs with dynamic batching in spark streaming, IEEE Transactions on Parallel & Distributed Systems) 1–1.
[26] A. Gounaris, J. Torres, Big Data Research 11 (2018) 22–32, https://doi.org/10.1016/j.bdr.2017.05.001.
[27] Y. Chen, J. Lu, C. Chen, M. Hoque, S. Tarkoma, Cost-effective resource provisioning for spark workloads. Proceedings of the 28th ACM International Conference on Information and Knowledge Management, CIKM 2019, 2019, pp. 2477–2480, https://doi.org/10.1145/3357384.3358090.Beijing, China, November 3-7, 2019.
[28] J. Liu, E. Pacitti, P. Valduriez, A survey of scheduling frameworks in big data systems, 2018, 7, 2, 103–128.
[29] Z. Tang, W. Ma, K. Li, K. Li, A data skew oriented reduce placement algorithm based on sampling, IEEE Transactions on Cloud Computing (99) (2016).1–1
[30] Q. Chen, C. Liu, Z. Xiao, Improving mapreduce performance using smart speculative execution strategy, IEEE Transactions on Computers 63 (4) (2014) 954–967.

[31] P.H. Ellaway, Cumulative sum technique and its application to the analysis of peristimulus time histograms, Electroencephalography & Clinical Neurophysiology 45(2)0–304.
[32] Z. Fu, Z. Tang, Optimizing speculative execution in spark heterogeneous environments, IEEE Transactions on Cloud Computing (2019), https://doi.org/10.1109/TCC.2019.2947674.1–1
[33] X. Huang, L. Zhang, R. Li, L. Wan, K. Li, Novel heuristic speculative execution strategies in heterogeneous distributed environments, Computers & Electrical EngineeringS0045790615002177.
[34] Open stack cloud operating system, https://www.openstack.org/.
[35] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, Kvm: the linux virtual machine monitor, 2007.
[36] Range partitioner, http://spark.apache.org/docs/latest/api/java/org/apache/spark/RangePartitioner.html.
[37] J. Yu, H. Chen, H. Fei, Sasm: Improving spark performance with adaptive skew mitigation. 2015 IEEE International Conference on Progress in Informatics and Computing (PIC), 2015.
[38] J. Lin, The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce (2012), 2009.
[39] Stanford large network dataset collection, http://snap.stanford.edu/data/index.htmlweb.
[40] Blog authorship corpus, http://u.cs.biu.ac.il/~koppel/BlogCorpus.htm.
[41] Twenty newsgroups, https://archive.ics.uci.edu/ml/datasets/Twenty+Newsgroups.