Future Generation Computer Systems ▮ (▮▮▮▮) ▮▮▮–▮▮▮

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

An intermediate data placement algorithm for load balancing in Spark computing environment

Zhuo Tang [a,*], Xiangshen Zhang [a], Kenli Li [a], Keqin Li [a,b]

[a] College of Information Science and Engineering, Hunan University, Changsha 410082, China

[b] Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

HIGHLIGHTS

- A novel sampling method for massive data as the input of Spark framework.
- An algorithm for filling the settled number of buckets with roughly equal number of tuples.
- Experiments are given to verify the performance and effects of the proposed algorithm.

ARTICLE INFO

Article history:
Received 30 November 2015
Received in revised form
12 May 2016
Accepted 23 June 2016
Available online xxxx

Keywords:
Data sampling
Data skew
Load balancing
MapReduce
Spark

ABSTRACT

Since MapReduce became an effective and popular programming framework for parallel data processing, key skew in intermediate data has become one of the important system performance bottlenecks. For solving the load imbalance of bucket containers in the shuffle process of the Spark computing framework, this paper proposes a splitting and combination algorithm for skew intermediate data blocks (SCID), which can improve the load balancing for various reduce tasks. Because the number of keys cannot be counted out until the input data are processed by map tasks, this paper provides a sampling algorithm based on reservoir sampling to detect the distribution of the keys in intermediate data. Contrasting with the original mechanism for bucket data loading, SCID sorts the data clusters of key/value tuples from each map task according to their sizes, and fills them into the relevant buckets orderly. A data cluster will be split once it exceeds the residual volume of the current bucket. After filling this bucket, the remainder cluster will be entered into the next iteration. Through this processing, the total size of data in each bucket is roughly scheduled equally. For each map task, each reduce task should fetch the intermediate results from a specific bucket, the quantity in all buckets for a map task will balance the load of the reduce tasks. We implement SCID in Spark 1.1.0 and evaluate its performance through three widely used benchmarks: *Sort*, *Text Search*, and *Word Count*. Experimental results show that our algorithms can not only achieve higher overall average balancing performance, but also reduce the execution time of a job with varying degrees of data skew.

1. Introduction

As a widely applied computing model for large-scale data processing, MapReduce can be used to parallelize the computation by running multiple map and reduce tasks over distributed data across multiple machines automatically and efficiently [1]. In current popular implementations of MapReduce [2,3], compared with Hadoop [4] and other distributed computing frameworks [5], Apache Spark has a more efficient implementation mechanism for large-scale data processing [6].

As the process of MapReduce in the Spark framework treats all the intermediate data as key/value tuples, a data cluster is the subset of all tuples with the same key [7]. Because mapper and reducer are the containers for map tasks and reduce tasks respectively [8], one way of implementations in Apache Spark is using hash algorithm to distribute the clusters to each reducer, and all clusters which are processed by the same reducer constitute a partition [9,10]. As the size of partitions depends on the number of relevant key/value tuples, data skew will give rise to the imbalance among different reducers because of the keys dispatching based on the hashing algorithm. As the skew of

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

http://dx.doi.org/10.1016/j.future.2016.06.027
0167-739X/

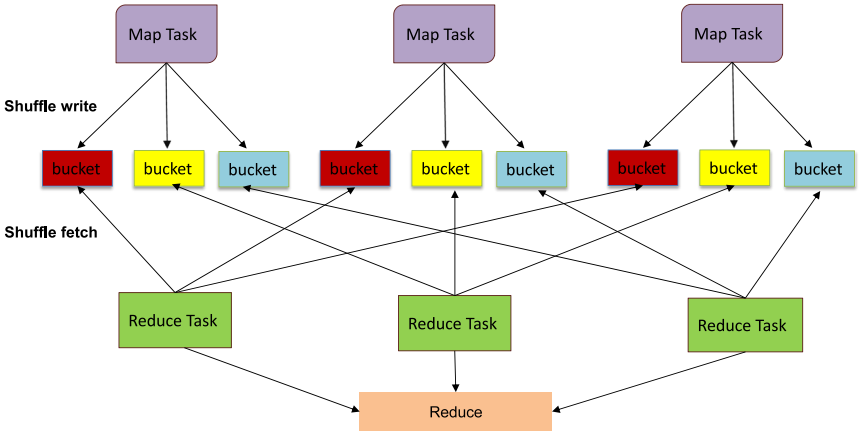


Fig. 1. The shuffle process in the Spark.

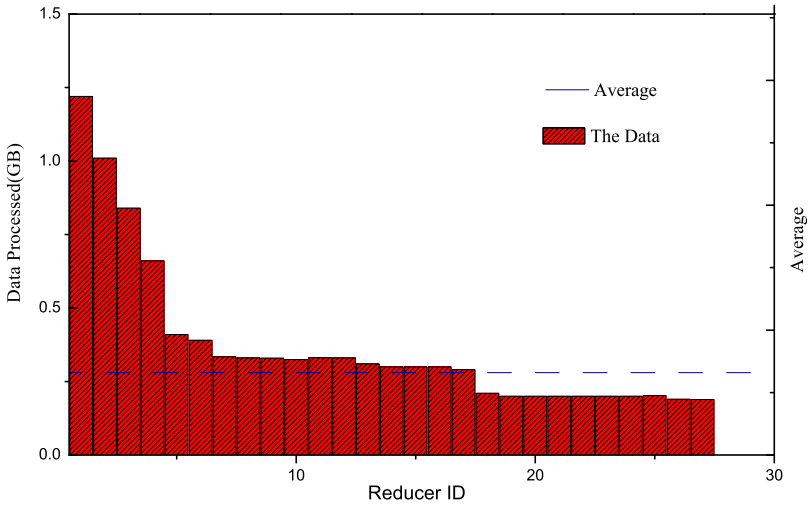


Fig. 2. Input data of each reducer in *Word Count* benchmark.

intermediate key/value tuples is universally existent in all input data [11,12], especially in the in-memory computing frameworks, data skew has become one of the main performance bottlenecks in the Spark distributed environment, which will cause the imbalance of the node workloads in the shuffle phase.

As shown in Fig. 1, reduce tasks have to wait until an arbitrary map task of a given job is finished.

In this framework, output are organized as data clusters which are transmitted between mapper and reducer, and a bucket is an array buffer for relevant reducer to collect the output of map tasks. In almost all literatures, data cluster is abbreviated as cluster. According to the number of reducers, each mapper will create the same number of buckets. That is, the number of buckets is $M \times R$, where M is the number of map tasks, and R is the number of reduce tasks.

It is clear that this computing framework cannot effectively deal with skewed data. For reduce tasks, partitioning skew will cause shuffle skew, which means some reducers will receive more data than others [13–15]. For example, Fig. 2 illustrates the different amounts of input data for each reducer when running the benchmark *Word Count* [16] using 10 GB of text data. For reduce tasks, partitioning skew will cause shuffle skew, in which some reduce tasks will receive more data than others. The results show that some reduce tasks suffer from many more inputs than others. In this situation, the task queue will arise on those reducers with heavy loads [17,18], and this will increase the completion time of running jobs, as well as degrade the system performance.

To cure the above problems, this paper proposes a splitting and combination algorithm for the skewed intermediate data (SCID) to implement efficient load balance among buckets in the Spark platform. As shown in Fig. 1, because each reducer always fetches data from a fixed bucket for each mapper, it is obvious that if we can keep the buckets for one mapper balance, the load balancing among all reducers could be improved. To make each bucket which collects the data from the same mapper assigned the equal size of data under its rated capacity, oversize clusters will be split, and the smaller ones will be combined to fill into the target buckets.

But the accurate policies of clusters adjustment are difficult to be generated if the intermediate keys distribution in the input data are unknown. In a real application, the intermediate outputs can be monitored and counted only after a job begins running, but it is meaningless to obtain the distribution of key/value tuples after processing all input data. To address this problem, this paper provides a dynamic range partition method that conducts a prerun sample of the input before the real job. By integrating sampling into a small percentage of the map tasks, this paper prioritizes the execution of sampling tasks over the normal Spark jobs to achieve the distribution statistics.

Finally, for the range partitioning modules in the benchmarks *Word Count* [16], *Sorting* [19], and *Text Search* [20] are sensitive to key skew, they are all appropriate to evaluate the performance of this proposed algorithm. The main contributions of this paper are summarized below.

- We apply the reservoir algorithm to implement the sampling for input data, and propose an evaluation model to select the

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

appropriate sample rate. This model can comprehensively consider the importance of cost, effect, and variance in sampling.

- We propose an algorithm to split and combine the clusters of key/value tuples. Through filling the settled number of buckets with roughly equal sized cluster combinations, it can receive better balance effect among the workloads of reduce tasks.
- Several groups of experiments are given to verify that the proposed algorithm can increase the performance of Apache Spark to deal with various degree of data skew.

The rest of the paper is organized as follows. Section 2 surveys related works on data skew in distributed processing environment. Section 3 introduces the overall system framework. Section 4 proposes the data sampling algorithm of the MapReduce framework. Section 5 proposes the algorithms of data cluster splitting and combination. The performance evaluation is given in Section 6. Section 7 concludes the paper.

2. Related work

As many MapReduce implementations have been used to process massive data, data skew is proved to be one of the most threats to the system performance. Afrati et al. implemented a multiway join as a single MapReduce process than as a cascade of 2-way joins [21]. They gave an algorithm to optimize the multiway join by minimizing the amount of replication of tuples from the input data. This algorithm can detect and fix the problems where an attribute is mistakenly included in the map keys, but still has the performance deficiencies when facing the data skew.

To optimize the performance in Hadoop framework, many algorithms and models about reduce tasks scheduling have been proposed in recent years. Hassan et al. proposed a MRFA-Join algorithm [22], which is a new frequency adaptive algorithm based on MapReduce programming model and a randomized key redistribution approach for join processing of large-scale data sets. To fix the problem of buffering all records from both inner and outer relations, Blanas et al. proposed a semi-join algorithm for log processing, and implemented an improved version of MapReduce sort-merge joins [23]. But these algorithms above still suffer from the baneful influence of data skew.

To solve data skew problems, several studies proposed some improved partitioning schemes based on the frequency of keys [24, 25]. Ibrahim et al. demonstrated that the presence of partitioning skew causes a huge amount of data transfer during the shuffle phase, and this will lead to significant unfairness on the reduce input among distributed data nodes [26]. To address this imbalance, they proposed a key partitioning algorithm named LEEN with awareness of locality and fairness. Gufler et al. defined a cost model that takes into account non-linear reducer tasks [27]. Based on the model, they proposed two load balancing approaches, fine partitioning and dynamic fragmentation that can deal with both skewed data and complex reduce tasks.

Yujie Xu et al. designed a MapReduce job to detect the frequencies of data inner keys, and estimated the overall distribution to make a partition scheme in advance [13]. Compared with these previous works, Gufler and other studies just directly fragmented and combined the partitions [28]. As a more fine-grained algorithm, our method can make the resized partitions be sent to the respective buckets by splitting and combination, based on a more accurate estimation for the intermediate data skew.

3. System overview

Fig. 3 shows the default distribution mechanism for data clusters in Spark 1.1.0 based on the hashing function. In this architecture, chucks are the data fragments, which is an organizational units of files with a default fixed size in HDFS. For

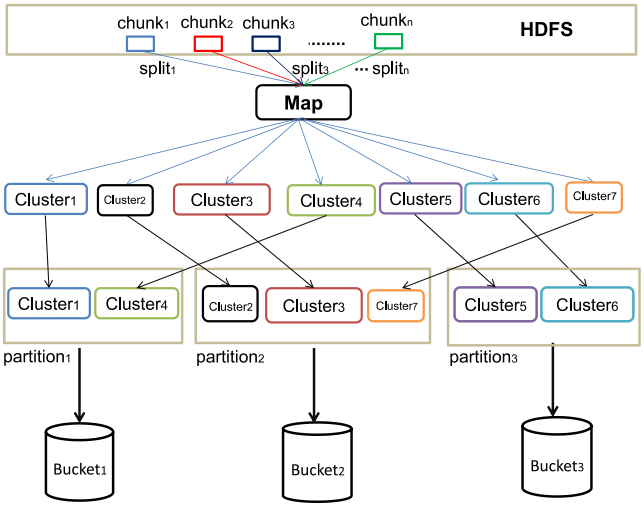


Fig. 3. Data distribution of shuffle in Spark 1.1.0.

map tasks, the input data are loaded into files in a distributed file system (HDFS) where each file consists of smaller chunks, which are called input splits. Each split is assigned to a map task. Map tasks process splits, and produce intermediate outputs which are usually partitioned and hashed to the relevant buckets. In this paper, we use $I \subseteq K \times V$ to represent the intermediate result from m map tasks, where K and V are respectively the sets of keys and values. According to the description in Section 1, a cluster is formalized as a subset containing all key/value tuples with a specific key k:

$$C_k = \{(k, v) \in I\}, \quad k \in K, v \in V. \tag{1}$$

In Fig. 3, the partition for intermediate tuples is determined by applying a partitioning function Π in Eq. (2):

$$\Pi : K \longrightarrow \{1, \ldots, p\}. \tag{2}$$

The intermediate results are split into p partitions according to the keys of the tuples. This way, all tuples belonging to the same cluster are placed into the same partition. A partition is thus a "input container" for a bucket which receives one or more clusters, and p can be as the number of buckets here. We can formalize a partition j as Eq. (3):

$$P(j) = \biguplus_{k \in K : \Pi(k) = j} C(k). \tag{3}$$

Fig. 4 represents an improved workflow of a Spark job, and the most critical component is the load balancing module. Before running Spark jobs, load balancer will generate a balanced partitioning strategy, which specifies how to split and combine the data clusters. This strategy can be used in the combination phase of shuffling process for a specific Spark job. In our proposed architecture, the load balancing module contains the following two phases.

Data sampling. For obtaining the treatment strategies for clusters in advance, it is necessary to ascertain the distribution of intermediate keys. Before a MapReduce computation begins with a map phase, where all the input splits are processed in parallel, a random sample of the required size will be produced. Data sampling is an independent and separate Spark job which running before the regular job in nature. Through the statistical numbers of keys in the sample data, we can forecast the rough sizes of all clusters which will be produced after map stages for the whole input. And this estimate can be as the direct input to generate the splitting strategies for the data clusters.

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

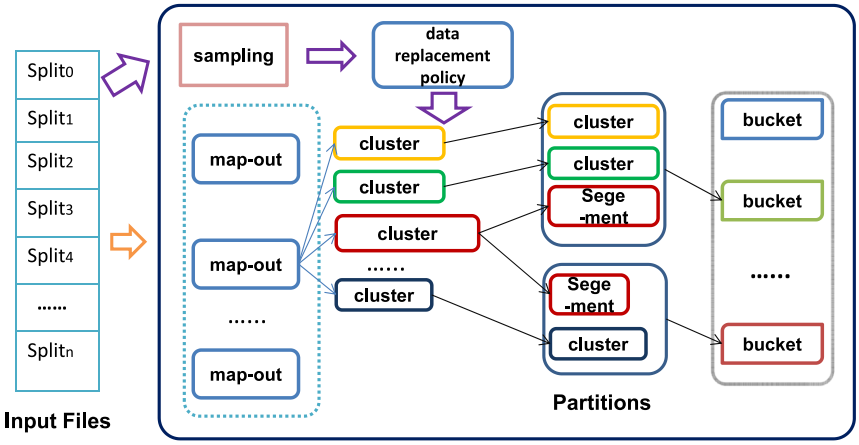


Fig. 4. Architecture with load balancing.

Splitting and combination. In this paper, for filling the target buckets equably as far as possible, some overlarge clusters should be split into several segments to fit the rated capacities of the buckets. This paper proposes a fine-grained splitting algorithm to eliminate the workloads of some reduce tasks caused by the cumbersome clusters. In this paper, splitting is just for combination: for a specific cluster, how to split is depend on the rated capacities of the target buckets.

The traditional assignment mechanism in the Spark framework lacks the perceptions to the inner data distribution. Based on the sampling, this paper proposes an improved computing framework to calculate the approximate proportions for different keys, which can be used as the basis of load-balanced partition strategies.

4. Data sampling in the Spark framework

4.1. Data skew model

In this model, for quantifying the sizes of the clusters received by a bucket with considering the effect of data skew, some initial and intermediate objects with their relationships can be formalized as follows.

As cluster is the collection of key/value tuples with a same key, the overall clusters can be formalized as a set C in Eq. (4):

$$C = \{C_1, C_2, \ldots, C_i, \ldots, C_m\}, \quad 1 \le i \le m \tag{4}$$

where m is the number of clusters. C_i is a structure, which can be formalized as $C_i = \{order, SC\}$, where $C_i.order$ records the initial order number of this cluster, and SC can be expressed as a separate sequence in Eq. (6).

And current buckets in the system can be formalized as a set B in Eq. (5):

$$B = \{B_1, B_2, \ldots, B_k, \ldots, B_n\}, \quad 1 \le k \le n \tag{5}$$

where n is the number of the buckets.

To record the number of key/value tuples in each cluster C_i, this model proposes a set SC to simplify this problem, which is shown in Eq. (6):

$$SC = \{SC_1, SC_2, \ldots, SC_i, \ldots, SC_m\}, \quad 1 \le i \le m \tag{6}$$

where SC_i is an integer which denotes the data size of a specific cluster.

More often than not, for the text data, the number of intermediate keys from original input data often follows the Zipf distributions [29,30]. In this model, we can use a varying parameter σ (from 0.1 to 1.2) to control the degree of skew, where larger value means heavier skew. In this paper, a matrix P is used to formalize this distribution. For a cluster may be split into different segments, $p_{k,i} \in P$ denotes the number of key/value tuples in a segment from cluster C_i which is dispatched to the bucket B_k. Hence, SC_i can be seen as the total size of segments in all buckets from cluster C_i as Eq. (7):

$$SC_i = \sum_{k=1}^{n} p_{k,i}. \tag{7}$$

In this model, the number of key/value tuples that the bucket k contains is denoted as $BC(k)$. For the key/value tuples distribution $p_{k,i}$ under a skew degree σ can be marked as $p^{\sigma}_{k,i}$, where σ is not a calculation parameter, and just reflects the skew degree of current input data. And the value of $BC(k)$ with skew degrees could be defined as Eq. (8):

$$BC(k, \sigma) = \sum_{i=1}^{m} p^{\sigma}_{k,i}. \tag{8}$$

On this basis, we can calculate the average number of $\langle key, value \rangle$ tuples of all buckets with current skew degree σ as Eq. (9):

$$\overline{mean_{\sigma}} = \frac{\sum_{k=1}^{n} BC(k, \sigma)}{n} = \frac{\sum_{k=1}^{n}\sum_{i=1}^{m} p^{\sigma}_{k,i}}{n} \tag{9}$$

where the parameter n is the number of buckets.

Naturally, the intermediate data processed by a bucket can be considered as skew using standard deviation. The clusters processed by a bucket are considered skew when the following condition in Eq. (10) is satisfied:

$$|\overline{mean_{\sigma}} - BC(k, \sigma)| > std \tag{10}$$

where σ denotes the current degree of data skew, and std is the standard deviation of all the buckets in $\langle key, value \rangle$ numbers, which can be used to measure the overall load balancing level of buckets. We can calculate the left part of Eq. (10) in the absolute value signs as Eq. (11):

$$\begin{aligned} \overline{mean_{\sigma}} - BC(k, \sigma) &= \frac{\sum_{k'=1}^{n}\sum_{i=1}^{m} p^{\sigma}_{k',i}}{n} - \sum_{j=1}^{m} p^{\sigma}_{k,j} \\ &= \frac{\sum_{k'=1}^{n}\sum_{i=1}^{m} p^{\sigma}_{k',i} - n\sum_{j=1}^{m} p^{\sigma}_{k,j}}{n}. \end{aligned} \tag{11}$$

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

Hence, the value of this standard deviation for all clusters in the buckets can be calculated by Eq. (12):

$$std(P,\sigma)=\sqrt{\frac{\sum_{k=1}^{n}\left(\frac{\sum_{v=1}^{n}\sum_{i=1}^{m}p_{v,i}^{\sigma}}{n}-\sum_{j=1}^{m}p_{k,j}^{\sigma}\right)^{2}}{n}}. \tag{12}$$

In this case, when a bucket is load balanced, $|\overline{mean_{\sigma}}-BC(k,\sigma)|<std$ is always satisfied. Because standard deviation is usually applied to reflect the range of fluctuations for a sequence, we can use the indicator *FoS* (factor of skew) to measure the load balancing degrees of all the buckets as Eq. (13):

$$FoS=std(P,\sigma)/\overline{mean_{\sigma}}. \tag{13}$$

Obviously, the smaller the value of *FoS*, the better the load balancing, and lower data the skew that will be obtained.

4.2. Reservoir sampling algorithm

Ascertaining the distribution of inner keys is the inevitable course to make the balancing of reducers. In a real application, the intermediate outputs can be monitored and counted only after a job begins running, but it is meaningless to obtain the key value distribution after processing all input data. Hence, calculating the optimal solution to the above problem is unrealistic, and the cost of pre-scanning the whole data will be hard to accept when the amount of data is very huge [31].

In this paper, for higher accuracy, we propose a reservoir sampling method to estimate the inner structure of large input data in the Spark framework. As a typical random sampling method, with the random data replacement policy in the sampling zone, this algorithm can achieve a much better approximation to the distribution of intermediate data. For sampling is with a small percentage of the input data, this paper prioritizes the execution of sampling job over the normal map tasks in order to achieve the distribution statistics.

Algorithm 1 Reservoir sampling

Require:
The *reservoir* size: r.
Ensure:
The sample data block set BS.
1: set BS as a set of the sample data block, $BS=\varnothing$;
2: //traverse blocks arriving from the input data
3: $k=0$;
4: **for** each block b in the input data **do**
5: k=k+1;
6: **if** $k\leq r$ **then**
7: //add the block b to the *reservoir*;
8: $BS=BS\bigcup b$;
9: **else**
10: select a block b' in the input data;
11: replace a randomly selected block in the *reservoir* with the block b';
12: **end if**
13: **end for**
14: **return** BS.

As shown in Algorithm 1, a uniform random sample with a fixed size k will be selected first without replacement from the input data, and the goal of this step is to form a *reservoir*. From $(k+1)$th chunk, the ith chunk will be selected with the probability: $1/i$ $(i=k+1,k+2,\ldots,N)$, and the chunk will appear in the *reservoir* with probability: k/i. In this process, an element in the original *reservoir* will be replaced randomly, until all of the nk chunks in *reservoir* are completely substituted. After one traversal, we can get entirely random k chunks from the original input data. It can be proved that each data will be taken out with equal probability k/n when the unknowable mount of all input data equals to n, $n\geq k$.

When $n=k$, the probability of each sample being taken out is equal according to the previous k samples put into *reservoir*, which is to say $k/k=1$. Set the current sample number to be n, so the probability of a number being taken out from the *reservoir* array is k/n. Then the problem becomes to prove that this theorem is also found in the situation of $n+1$: the probability of putting $(n+1)$th element into the *reservoir* is $k/(n+1)$. For a random chunk in the previous n elements we have $(k+1)\leq m\leq n$. First, the model can be defined simply as follows:

(1) The selection probability of mth chunk: k/m;
(2) The selection probability of $(m+1)$th chunk: $k/(m+1)$;
(3) The probability that the mth chunk is not replaced by $(m+1)$th: $1-1/k$;
(4) The probability that the $(m+1)$th chunk is not selected: $1-k/(m+1)$. Hence, the equation to calculate the probability that mth chunk appears in the *reservoir* is as follows:

$$\begin{aligned}P(m)&=\frac{k}{m}\times\left\{\left[\left(\frac{k}{m+1}\right)\times\left(1-\frac{1}{k}\right)+\left(1-\frac{k}{m+1}\right)\right]\times\cdots\right.\\&\quad\left.\times\left[\left(\frac{k}{n+1}\right)\times\left(1-\frac{1}{k}\right)+\left(1-\frac{k}{n+1}\right)\right]\right\}\\&=\frac{k}{m}\times\left[\frac{m}{m+1}\times\frac{m+1}{m+2}\times\cdots\times\frac{n}{n+1}\right]\\&=\frac{k}{m}\times\frac{m}{n+1}\\&=\frac{k}{n+1}.\end{aligned} \tag{14}$$

Eq. (14) clearly proves that the probability of each sample being taken is equal in the situation of $n+1$. Conventional uniform sampling will inevitably result in a certain number of multi-duplicated samples. Because all random number generators in the Java and Scala languages are simply pseudorandom functions, for large scale data, especially with increasing sampling space, they cannot guarantee that all sample data are completely randomized. From the description in Algorithm 1, the main process of reservoir sampling is to save k preceding elements first (k is the sample number and also the size of the reservoir) and then randomly replace original selected elements in the reservoir using a new element that is selected from outside the reservoir in a different probability. The final k sample data will be generated after finishing the traversal for current input data. Compared with pseudorandom functions, reservoir sampling can ensure randomness, especially when taking the data from some sequence flows, and it is ideal for reading the input data from large texts line by line in the Spark framework. Compared with conventional uniform sampling, reservoir sampling can ensure that the key distributions are closer to the whole situation in the original data.

4.3. Prediction of cluster sizes

Algorithm 2 provides the process to evaluate the data size for each cluster.

For a specific Spark job, this algorithm first starts this job with the sample data, and records the size of clusters from a map local node based on a monitor. From Algorithm 1, based on a basic assumption that the distribution probability of keys in sample data mostly keep coherent with all input data, we can roughly estimate the data size of each cluster from every map node.

Algorithm 2 illustrates the whole process for cluster size estimation. First, for a specific Spark job, the input data set BS is the sample of data blocks as the output of Algorithm 1, and mr_{job} represents the Spark job for sampling. For the sampled data, Algorithm 2 counts the number of clusters based on a monitor deployed on map nodes, and obtains a set $SC=\{SC_i\}$, where SC_i

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

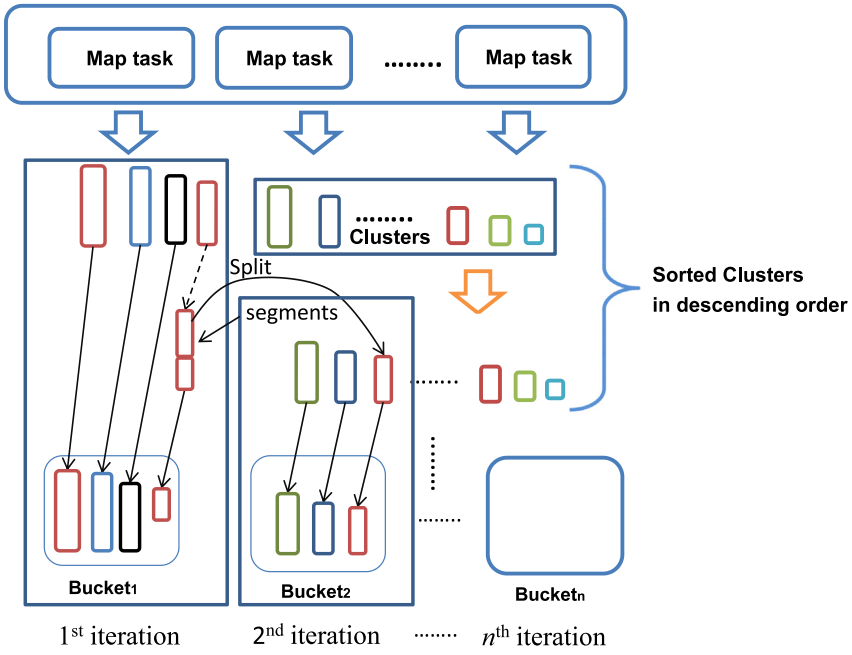


Fig. 5. The steps of clusters distribution.

Algorithm 2 Cluster Size Prediction

Require:
The sample of data blocks BS;
A MapReduce job: mr_{job}.
Ensure:
SC: The number set of clusters for overall input data.
1: initialize each element in SC: $SC_i = 0$;
2: run the Spark job mr_{job} using BS as the input data;
3: get a tuples set $MOUT$ from the map out of Spark job mr_{job};
//initialize the number for each tuples with a same key
4: initialize the counter $i = 0, j = 0$;
5: sort the tuples $< k, v >$ in $MOUT$ on keys;
6: count the number of set $MOUT$: N;
//count the number for tuples with a same key
7: **while** $j \neq N$ **do**
8: get the key K_r of the $r - th$ tuple;
9: $m = 1$;
10: **for** $r = j; r < N$; **do**
11: **if** $K_{r+1} == K_r$ and $r! = N - 1$ **then**
12: $m = m + 1$;
13: **else**
14: $SC_i = m$;
15: $i++$;
16: $j = r$;
17: break;
18: **end if**
19: $r++$;
20: **end for**
21: SN = the number of tuples in BS;
22: WN = the number of tuples in all input data;
23: $SC_i = (SC_i \times WN) \,/\, SN$;
24: **end while**
25: produce the set of clusters C from $MOUT$;
26: **return** SC.

denotes the number of $\langle k_i, value \rangle$ tuples for the sampled data. Because the number of key/value tuples for each key in sample set is usually in proportion to key number of the original data roughly, we can estimate the cluster size by scaling up the approximate number of each cluster from sample data.

5. Cluster splitting and combining

5.1. Process description

In this section, a bucket packing algorithm by splitting the clusters is proposed to achieve the data balancing. In this algorithm, we record the rated capacity of each bucket as the average W_{avg}, which can be calculated as Eq. (15):

$$W_{avg} = \frac{\sum_{i=1}^{m} SC_i}{n} \tag{15}$$

where m is the number of clusters, and n is the number of buckets. With this upper limit for containing clusters, the current residual volume array of buckets can be denoted as: $\{RB_1, RB_2, \ldots, RB_j, \ldots, RB_n\}$.

As shown in Fig. 5, at the beginning of Algorithm 3, the residual volumes of buckets at the first time are all initialed as W_{avg}. For the first iterator, we sort the set $\{C_i\}$ in ascending order according to the cluster sizes. From the maximal cluster C_m, if $SC_m \geq RB_1$, then a new segment will split out from C_m with size W_{avg} to be dispatched into B_1. And only the remaining part of C_m with size of $SC_m - RB_1$ and all remaining clusters need to enter into the next iterator. In this case, we just use one step to fill a bucket. But in most cases, it usually cannot fill any bucket with the current maximal cluster, that is: $SC_m < RB_1$. Therefore, we need to put C_m into the first bucket. For the remaining space, the current second maximal C_{m-1} will be checked to see whether it can fill the space. If $SC_m + SC_{m-1} \geq RB_1$, then cluster C_{m-1} will be split, else cluster C_{m-1} will be dispatched to this bucket, and the process will traverse all the residual clusters forward until a cluster C_i satisfy this condition:

$$\sum_{j=m}^{i} C_j \geq RB_k, \quad (j = m, m-1, \ldots, i) \tag{16}$$

where k is the sequence number of current bucket. In this algorithm, because it will fill up a bucket in each iteration step, the number k can also denote the current iteration times. And in each iterator, the set of clusters size SC will be resorted. Through this processing, the total sizes of data from a specific mapper for all buckets are scheduled roughly equal.

5.2. Algorithms

Algorithm 3 illustrates the overall process, for shuffle process will start once a definite proportion of map tasks output the

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

intermediate data (in spark 1.1.0, the default ratio is 20%), all elements in clusters $\{C_1, C_2, \ldots, C_i, \ldots, C_m\}$ cannot be coexist at the same time. But this situation will not affect the feasibility of our algorithm, because we can estimate the overall cluster sizes from the sampling process, after running the sampling algorithm before running the actual job, Algorithm 3 is indeed a process simulator: it just need to output a data placement policy by simulating the processing of cluster splitting and combination instead of moving the practical data. Moreover, a simple bubble sorting "*ClusterSORT*" are illustrated in Algorithm 4, which is as a subprocedure of Algorithm 3.

Algorithm 3 Splitting and Combining of Clusters

Require:
$C = \{C_1, C_2, \ldots, C_j, \ldots, C_m\}$;//The set of tuples clusters
$SC = \{SC_1, SC_2, \ldots, SC_i, \ldots, SC_m\}$;//The number set of each cluster in set C
$B = \{B_1, B_2, \ldots, B_k, \ldots, B_n\}$; //The current buckets list
$RB = \{RB_1, RB_2, \ldots, RB_k, \ldots, RB_n\}$.//The current residual volume array of bucket
Ensure:
//A pre set data placement policy
Matrix P: $P = \{p_{i,j}\}$, $1 \le i \le m$, $1 \le j \le n$.
1: Initial Matrix P: $p_{i,j} = 0$;
2: $W_{avg} = 0$;
3: **for** $i = 1; i \le m; i++$ **do**
4: $W_{avg} = W_{avg} + SC_i$;
5: **end for**
6: $W_{avg} = W_{avg}/n$;
7: **for** $k = 1; k \le n; k++$ **do**
8: $RB_k = W_{avg}$;
9: **end for**//for each bucket
10: **for** $k = 1; k \le n; k++$ **do**
11: //sort current remaining cluster by SC in descending order, using Algorithm 4
12: $C = ClusterSORT(C)$
//for each clusters, from the max one
13: **for** $i = 1; i \le m; i++$ **do**
14: **if** $SC_i \le RB_k$ **then**
15: $RB_k = RB_k - SC_i$
//put C_i into bucket B_k;
//j is the initial order number of C_i before sorted.
16: $j = C_i.order$;
17: $p_{k,j} = SC_i$;
18: **else**
19: break; //Find a cluster C_i large than the remaining space of current bucket B_k
20: **end if**
21: **end for**
22: $SC_i = SC_i - RB_k$; // Split cluster C_i, fill bucket B_k
23: **end for**
24: **return** P.

Algorithm 3 is actually an application of First Fit Decreasing (FFD) solution for bin-packing problem. For the sizes of bins and items are known in advance, as a greedy algorithm, FFD is proven an effective method which can get the locally optimal solution quickly. And in our case, because the items (data clusters) can be split and recombined, FFD can obtain an almost perfect result to fit the buckets with their preset capacities. And because the current remainder clusters need to be reordered in this algorithm, but the matrix P is used to record the distribution polices of the original data clusters, as lines 15–17 in Algorithm 3, the structure $C_i.order$ is used to record the original order number of the residual cluster in this process.

Eq. (17) shows the matrix P output from Algorithm 3 as a preset of data placement polices:

$$P = \begin{bmatrix} p_{1,1} & p_{1,2} & p_{1,3} & \cdots & p_{1,j} & \cdots & p_{1,m} \\ p_{2,1} & p_{2,2} & p_{2,3} & \cdots & p_{2,j} & \cdots & p_{2,m} \\ & & \cdots & \cdots & \cdots & & \\ p_{i,1} & p_{i,2} & p_{i,3} & \cdots & p_{i,j} & \cdots & p_{i,m} \\ & & \cdots & \cdots & \cdots & & \\ p_{n,1} & p_{n,2} & p_{n,3} & \cdots & p_{n,j} & \cdots & p_{n,m} \end{bmatrix}. \quad (17)$$

Algorithm 4 ClusterSORT

Require:
//The set of tuples clusters
$C = \{C_1, C_2, \ldots, C_j, \ldots, C_m\}$;
//The number set of each cluster in set C
$SC = \{SC_1, SC_2, \ldots, SC_i, \ldots, SC_m\}$.
Ensure:
The set of sorted clusters: C'.
1: **for** $j = 1; j \le m; j++$ **do**
2: **for** $i = 1; i < m - j; i++$ **do**
3: **if** $SC_i < SC_{i+1}$ **then**
4: $temp = SC_i$;
5: $SC_i = SC_{i+1}$;
6: $SC_{i+1} = temp$;
7: $tempC = C_i$;
8: $C_i = C_{i+1}$;
9: $C_{i+1} = tempC$;
10: **end if**
11: **end for**
12: **end for**
13: **return** $C' = C$.

As the definition in Section 4.1, data placement matrix $P = \{p_{i,j}\}$ denotes the number of key/value tuples from the jth cluster which should be dispatched to the ith bucket. This matrix can be expanded as a set of tuples sets, which can directly denotes the number of processed key/value tuples from every cluster for each bucket as Eq. (18):

$$\begin{cases} B_1, \{\langle k_1, p_{1,1}\rangle, \langle k_2, p_{1,2}\rangle \cdots \langle k_j, p_{1,j}\rangle \cdots \langle k_m, p_{1,m}\rangle\} \\ B_2, \{\langle k_1, p_{2,1}\rangle, \langle k_2, p_{2,2}\rangle \cdots \langle k_j, p_{2,j}\rangle \cdots \langle k_m, p_{2,m}\rangle\} \\ B_3, \{\langle k_1, p_{3,1}\rangle, \langle k_2, p_{3,2}\rangle \cdots \langle k_j, p_{3,j}\rangle \cdots \langle k_m, p_{3,m}\rangle\} \\ \cdots \\ B_i, \{\langle k_1, p_{i,1}\rangle, \langle k_2, p_{i,2}\rangle \cdots \langle k_j, p_{i,j}\rangle \cdots \langle k_m, p_{i,m}\rangle\} \\ \cdots \\ B_n, \{\langle k_1, p_{n,1}\rangle, \langle k_2, p_{n,2}\rangle \cdots \langle k_j, p_{n,j}\rangle \cdots \langle k_m, p_{n,m}\rangle\}. \end{cases} \quad (18)$$

Algorithm 5 presents how to dispatch the intermediate data from map out to each bucket, where the matrix P is used as an input parameter. We use a matrix $CB = \{CB_{i,j}\}$ to formalize the current loads of all buckets, in which $CB_{i,j}$ denotes the current number of key/value tuples from jth cluster which has been dispatched to ith bucket. At the beginning of algorithm, this matrix should be initialed as $CB_{i,j} = 0$ for all clusters and buckets.

Algorithm 5 can implement the distribution of the intermediate data under the results of the above Algorithms 1–4. Once a map outputs a tuple $\langle K_i, v\rangle$, the target of this algorithm is to determine which bucket should be chosen to accept this key. As the input of Algorithm 5, matrix P can be acquired from Algorithm 3, whose input contains a parameter C, which denotes the set of sizes for each cluster among all input data, but it is an estimated value from Algorithm 2 based on the sampling of the overall input data. From the above analysis, we can reach the conclusion that there maybe some keys are not contained in the clusters set C. That is to say, we should judge whether this tuple $\langle K_i, v\rangle$ can be dispatched under matrix P.

For the clusters set C is one of output from Algorithm 3 which being sorted in descending order, Algorithm 5 first travel C to get the location of key K_i. If there is not any key/value tuple in clusters set C can match this key K_i, this tuple should be dispatched under the default hashing function. Otherwise, we can dispatch this key under the current matrix P, which denotes the maximum number of each key for every bucket. The actual implementation is to travel the relevant column vector $\overrightarrow{p_j}$ in matrix P. The ordinal number of the column is up to the location of the key K_i in clusters set C. For vector $\overrightarrow{p_j}$, every element denotes the allowed maximal quantities of the tuples with this key K_i in each bucket.

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

Table 1
The software and hardware configurations in the Hadoop cluster.

The node type	Master	Slave
Software environment	ubuntu 12.04, JDK 1.7 Hadoop 2.6.0, Spark 1.1.0	ubuntu 12.04, JDK 1.7 Hadoop 2.6.0, Spark 1.1.0
CPU	4 cores, 2.7 GHz	4 cores, 2.7 GHz
Memory	8G	8G
Quantity	1	15

Algorithm 5 Cluster Dispatching

Require:
A MapReduce job: MR_{job};
The number of clusters: m;
The number of buckets: n;
//The set of tuples clusters
$C = \{C_1, C_2, \ldots, C_j, \ldots, C_m\}$;
Matrix P: $P = \{p_{i,j}\}$, $1 \le i \le n$, $1 \le j \le m$.
Ensure:
Dispatch a tuple $\langle key, value\rangle$ from output of map tasks to appropriate buckets.
//initialize all cluster number in all buckets
1: $CB_{i,j} = 0$, $1 \le i \le n$, $1 \le j \le m$;
2: run the Spark job MR_{job};
3: start receive the tuples set $T = \langle k, v\rangle$ from the map out of Spark job mr_j;
4: get the key k_i of the $i-th$ tuple $\langle k_i, v\rangle$;
//get the order no j of the key k_i in clusters C
5: **for** $j = 1; j \le m; j++$ **do**
6: **if** k_i == the key of C_j **then**
7: break;
8: **end if**
9: **end for**
10: $j = C_j.order$;
//travel each row of the matrix P
11: **if** ($j \ne m$) or ($j == m$ and k_i == the key of C_m) **then**
12: **for** $i = 1; i \le n; i++$ **do**
13: **if** $CB_{i,j} \le p_{i,j}$ **then**
14: dispatch the tuple $\langle k_i, v\rangle$ to bucket B_i;
15: $CB_{i,j}++$;
16: **end if**
17: **end for**
18: **else**
19: dispatch the tuple $\langle k_i, v\rangle$ under the default hashing function;
20: **end if**
21: **return** Success.

This algorithm travels each element of $\overrightarrow{p_j}$ to judge whether current $CB_{i,j}$ is less than the limit $p_{i,j}$. The actual meaning is as follows: for key K_i, this solution will travel all buckets in this system. If current bucket B_i is not full for this key, then dispatch the tuple $\langle k_i, v\rangle$ to this bucket. Because the number of various keys in matrix P (which also means the column number in P) is usually less than the actual number of keys for the incomplete sampling, from Eq. (15), it is easy to know that the up limit numbers of each key for every bucket should be larger than the actual distributed key numbers. Obviously, the incomplete sampling is one of the main reasons of the non-completely balance among the bucket workloads.

6. Evaluation

6.1. Experiment setting

SCID has been evaluated on a practical test cluster, which includes 15 slave nodes and 1 master node connected by a Gigabit Ethernet switch. The experimental environment is based on the Spark 1.1.0, and the hardware and software configurations are shown in Table 1. All experiments use the default configurations in Spark for HDFS.

In the following experiments, to verify the advantages of SCID, we evaluate this model for *FoS* and the execution time using these common benchmarks: *Sort*, *Text Search*, and *Word Count*. To estimate the defects and advantages of this proposed algorithm SCID objectively, the following compared algorithms are chosen to be implemented with the above benchmarks in the same experiment environment.

Range: range partition. This is a widely used algorithm of partition distribution. In this method, the intermediate $\langle key, value\rangle$ tuples are sorted by key first, and then the tuples are assigned to reduce tasks according this key range sequentially. Range can improve the data balance among reduce tasks to a certain degree. But its poor locality in the reduce phase and the redundant inner communication finally results in the longer execution time [32].

LIBRA: lightweight implementation of balanced range assignment. LIBRA is a lightweight strategy to solve the data skew problem for reduce-side applications in MapReduce. It uses an innovative sampling method which can achieve a highly accurate approximation to the distribution of the intermediate data by sampling only a small fraction of the intermediate data during the normal map processing [14].

PCWC: partition combination & *Word Count*. The algorithm is in order to obtain a better load balancing level when the degree of data skew is serious. The basic idea is to split the large clusters into equal number of pieces according to the quantity of reduce tasks. With large data skew, there are a great differences among the cluster sizes, and PCWC can hardly distribute the $\langle key, value\rangle$ tuples to reduce nodes evenly [13].

DEFH: default hashing algorithm. It is the default mechanism in the Spark computation environment which can obtain a good performance only when keys equally appear with an uniformly distribution.

The performance of different reduce placement algorithms are compared through two key indicators with the goal of measuring the impact of intermediate data replacement. (1) Execution time. For the main effect of SCID is to balance the workload of the reduce tasks, it can decrease the length of the longest tasks waiting queue effectively. For a specific benchmark, this performance index can reflect the advantages of the proposed algorithm. As similar with Hadoop, in the Spark 1.1.0 framework, the start time and the finish time of map or reduce processing can also be measured directly. (2) Load balancing degree. We use the model *FoS* to quantize the load balancing among the all buckets in the Spark computation framework directly. In these experiments, we design and implement a monitor in Spark 1.1.0 to measure and record the current data size for the all buckets, and these observed values can be used as the input parameters of *FoS*.

6.2. Performance evaluation

6.2.1. Sampling experiments

In this section, we first propose an evaluation formula as Eq. (19) to select the appropriate sample rate, which can comprehensively consider the importance of cost, effect, and variance in sampling.

$$i = argMin[f_i(\Delta_i, T_i, \Phi_i) = \alpha\Delta_i + \beta T_i + \gamma\Phi_i] \tag{19}$$

where function $f_i(\Delta_i, T_i, \Phi_i)$ is a comprehensive index considering both cost and effect, in which Δ_i reflects the difference among the sequences of *FoS* values between the currently adopted percentage

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

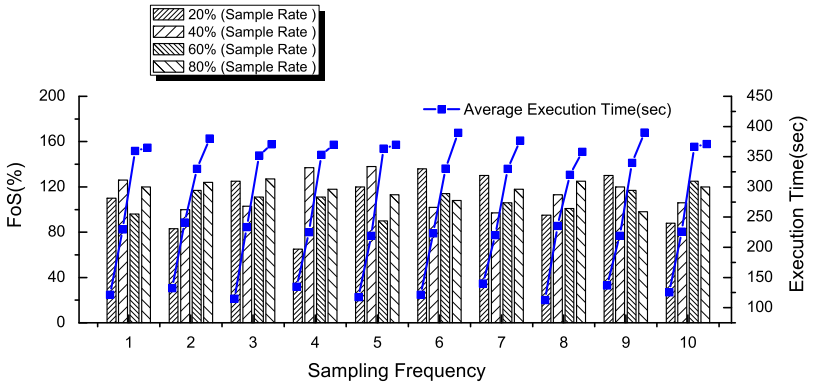


Fig. 6. The comparison experiment in various sampling rate.

Table 2
FoS values of each sampling.

i	Sample rate (j)	1	2	3	4	5	6	7	8	9	10
1	20%	110	83	125	65	120	136	130	95	130	88
2	40%	115	92	102	128	127	98	97	113	115	98
3	60%	96	117	111	111	90	114	106	101	117	125
4	80%	120	124	127	118	113	108	118	125	98	120

Table 3
Execution time of each sampling.

i	Sample rate (j)	1	2	3	4	5	6	7	8	9	10
1	20%	121.5	132.5	115	134.5	117.5	121.5	140	112.5	137	125.5
2	40%	229.5	241	233.5	225	219	223	220	235.5	219	226
3	60%	359.5	330	352	353	363.5	330.5	330	320	340	366.5
4	80%	365	380	370.5	370	370	389.5	376.5	358	390	371

and 100% (the whole input data set), which can be calculated by Eq. (20):

$$\Delta_i = \sqrt{\sum_{j=1}^{N}\left(d_{i,j} - \frac{\sum_{r=1}^{N} d_{4,r}}{N}\right)^2} \tag{20}$$

where N denotes the experimental repetition times, and $d_{i,j}$ represents the *FoS* value obtained in the jth sampling experiment under the ith sampling rate. $1 \le i \le SN$ is the order number of different sampling percentages: {20%, 40%, 60%, 80%}, and SN denotes the space size of different sampling rates. For this experiment, $SN = 5$, and $d_{4,j}$ denotes the values with a 80% sampling rate. As an average sampling execution time, T_i can be calculated simply by Eq. (21):

$$T_i = \frac{\sum_{j=1}^{N} t_{i,j}}{N} \tag{21}$$

where $t_{i,j}$ represents the execution time of the jth sampling experiment under the ith sampling rate, $1 \le i \le SN$ and $1 \le j \le N$.

To full consider the influence of data volatilities, Eq. (22) provides the process to calculate the parameter Φ_i based on the standard deviation formula:

$$\Phi_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N}\left(d_{i,j} - \frac{1}{N}\sum_{j=1}^{N} d_{i,j}\right)^2}. \tag{22}$$

Fig. 6 shows the *FoS* and execution time obtained in times sampling experiments. The original records about *FoS* values and execution time in Fig. 6 are as shown in Tables 2 and 3 respectively.

Table 4
Comprehensive evaluation for cost and effect with different benchmarks.

i	Rate	Δ_i	T_i	Φ_i	f_i
1	20%	77.61	125.75	22.87	226.23
2	40%	40.36	227.15	12.22	279.73
3	60%	41.33	344.5	10.10	395.93
4	80%	26.28	374.05	8.31	408.64

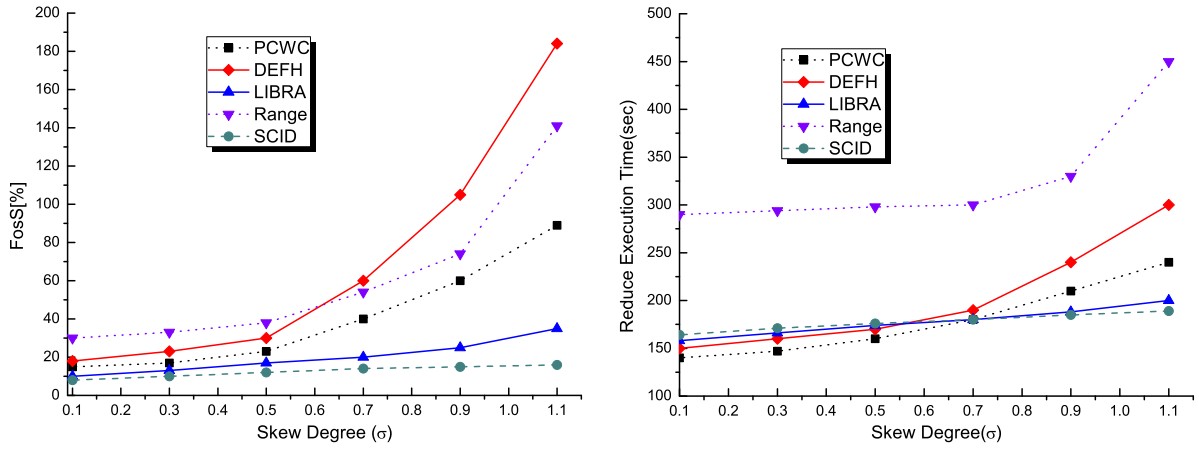
Table 4 provides the final values of Δ_i, T_i, and Φ_i for all benchmarks with various sampling rates. Each group of sampling experiments is repeated ten times, which means that the parameter N in Eqs. (17)–(19) should be set to 10 in these experiments. Finally, for the weight coefficients, we can simply set $\alpha = \beta = \gamma = 1$. That is, we think that the cost, effect, and data volatility are equally important. According to Eq. (19), it is easy to learn that sampling 20% of the map tasks is an appropriate choice for the input data.

6.2.2. Sort benchmark testing

Fig. 7 shows that the reduce execution time of the above five algorithms all increase with the data skew degree varied from 0.1 to 1.1 under the *sort* benchmark. Fig. 7(a) represents the variation of load balancing when the degrees of data skew ranges from 0.1 to 1.1. It is obvious that the value of *FoS* increases rapidly when the degree of skew exceeds 0.65 for all experimental algorithms. Through the clusters partition and placement, SCID can make the sizes of processed data more even in reduce tasks.

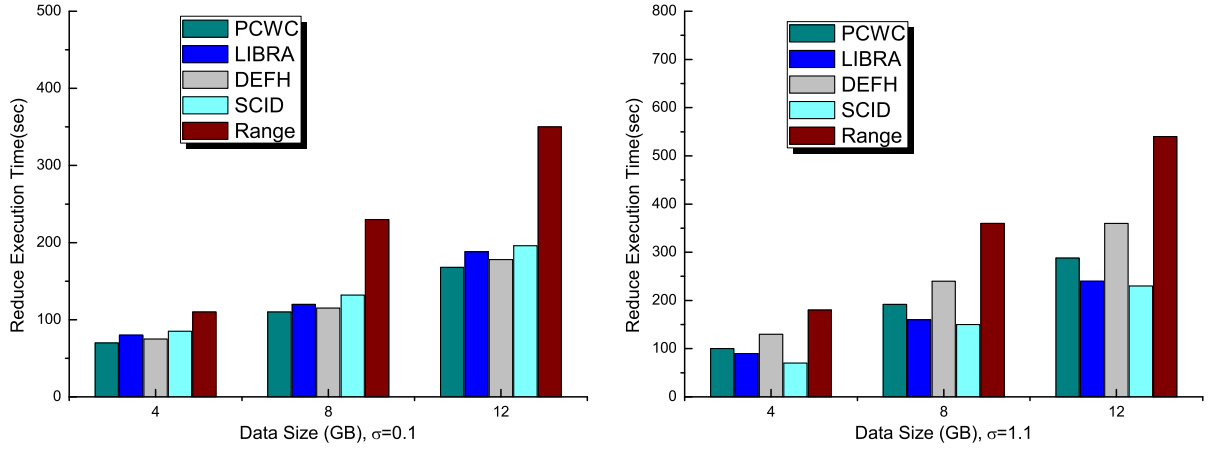
Fig. 7(b) shows the effects to the system performance for the imbalance of data size among different reducers. The starting points of the reduce tasks are identified as the time once a definite proportion of map tasks output the intermediate data (in our Spark platform, the default ratio is 20%). It is easy to see that with the lower skew degree, PCWC, DEFH, and LIBRA are slightly better than SCID. This is because although the processing of generating the

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027



(a) Load balancing. (b) Reduce execution time.

Fig. 7. Performance vs. data skew degrees for *Sort*.



(a) $\sigma = 0.1$. (b) $\sigma = 1.1$.

Fig. 8. Reduce execution time vs. data size for *Sort*.

distribution strategy matrix P which are described in Algorithms 1–4 are completed in a previous sampling job, compared with the normal reduce tasks which running on the Spark framework, there are still a degree of additional computing for SCID to get the order number of each key in strategy matrix P to determine how to dispatch the $\langle key, value \rangle$ pairs, which is just the process of algorithm 5. Compared with other algorithms, the method proposed in Algorithm 5 which determines the distribution of the intermediate data is actually tuple by tuple. Although this way can bring more fine-grained control in intermediate data distribution, the time cost is significant.

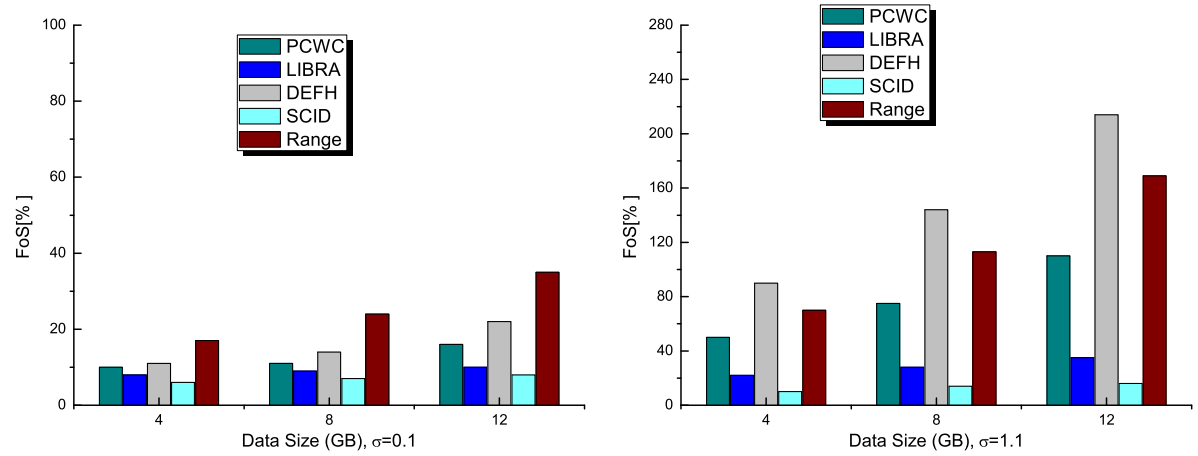
But when skew degree is larger than 0.7, the performance can be improved greatly by the optimized placement policies for the intermediate data. By balancing the workloads for the reduce tasks, the influences of the extra operations can be eliminated to a much greater extent. It is obvious that the execution time of reduce tasks under SCID increases much slower than other algorithms. Fig. 8(a) reflects the same phenomenon. And for these reasons, the similar situations happen in the experiments when using the benchmark *Word Count*.

Fig. 8 represents the variation of reduce execution time when the data size ranges from 4 GB to 12 GB under two different skew degrees ($\sigma = 0.1$ and $\sigma = 1.1$). When the data set has a lower skew degree (see Fig. 8(a)), PCWC has the highest time-efficient, and LIBRA and SCID perform worse than DEFH because the extra overheads caused by cluster splitting and combination. We have to admit that because the performance improvements are not apparent by balancing the workload of the inner tasks, the negative effect caused by the extra cluster adjustments to the overall performance is significant.

However, Fig. 8(b) highlights the effect of the optimizations from SCID with heavy data skew. The reduce execution time of SCID has always been the minimum with the skew degree $\sigma = 1.1$. And as the data size increases, due to the performance improvement by data balancing mechanism, the growth rate of execution time under SCID is smaller than others. This experimental results demonstrate that SCID can achieve better performance with heavy skew degree. The larger size of data, the more improvement can be attained.

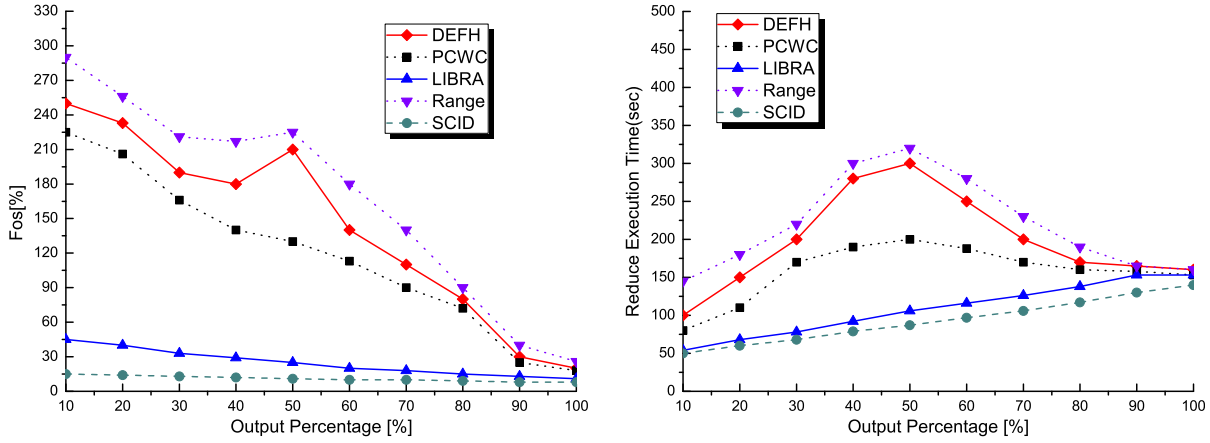
Fig. 9(a) shows the experimental results with light data skew ($\sigma = 0.1$), and Fig. 9(b) is with the heavy data skew ($\sigma = 1.1$) relatively. It is obvious that the buckets balance will become worse with more skew input data. When the data size ranges from 4 GB to 12 GB with a lower skew degree, it is can be observed that although the *FoS* values of all algorithms increase slowly, but this indicator of SCID is always better than other compared algorithms. As the data set scale increases rapidly, the performance of SCID is more prominent: the *FoS* values with SCID is 40% smaller than DEFH in the Spark implementation. As shown in Fig. 9(b), the balance degrees for all reduce tasks with SCID are all less than 15%, whereas, it reaches to 100%, 25%, 200%, and 150% with PCWC, LIBRA, DEFH, and Range, respectively.

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027



(a) $\sigma = 0.1$. (b) $\sigma = 1.1$.

Fig. 9. Load balancing vs. data size for *Sort*.



(a) Load balancing. (b) Execution time.

Fig. 10. Performance vs. query percentages for *Text Search*.

6.2.3. *Text search benchmark testing*

In this experiment, SCID and all compared algorithms are evaluated using *Text Search* benchmark with an English Wikipedia archive data set [33]. *Text Search* is a widely used benchmark in the Spark framework, whose realization mechanism is similar to *Grep* benchmark in the Hadoop. Because the behavior of *Text Search* depends on how frequently the search expression appears in the input file, we tune the search expression and make the output percentages vary from 10% to 100% of the input.

From Fig. 10(a), we can learn that SCID performs more efficiently to balance the buckets workloads than other algorithms especially with lower query percentages. In fact, with less proportions of text data being searched, it will be easy to cause serious skew for the intermediate key/value tuples. In this situation, it will cause imbalance among different buckets, and degrade the system performance during the reduce phase. When the output percentage is higher, because the resulting data become more evenly distributed, the performance difference becomes smaller, although SCID is still slightly better.

A similar trend is also appearing in Fig. 10(b), with more balanced data in the fixed number of buckets, reduce task has a higher efficiency to process the intermediate data in parallel. When the output percentage equal to about 50%, the maximum values of these curves mean that DEFH and Range both have worse workload balancing in buckets. This is because that when the size output results are close to half of the input data, the distributions of intermediate keys are often most uneven, which will cause the bad balance and poor system performance.

Fig. 11 shows the variation of reduce execution time when the data size ranges from 10 to 30 GB with different query percentages. From these results, we can conclude that the executing time increases more slowly contrasting with the growth of the data size, especially for the first four algorithms.

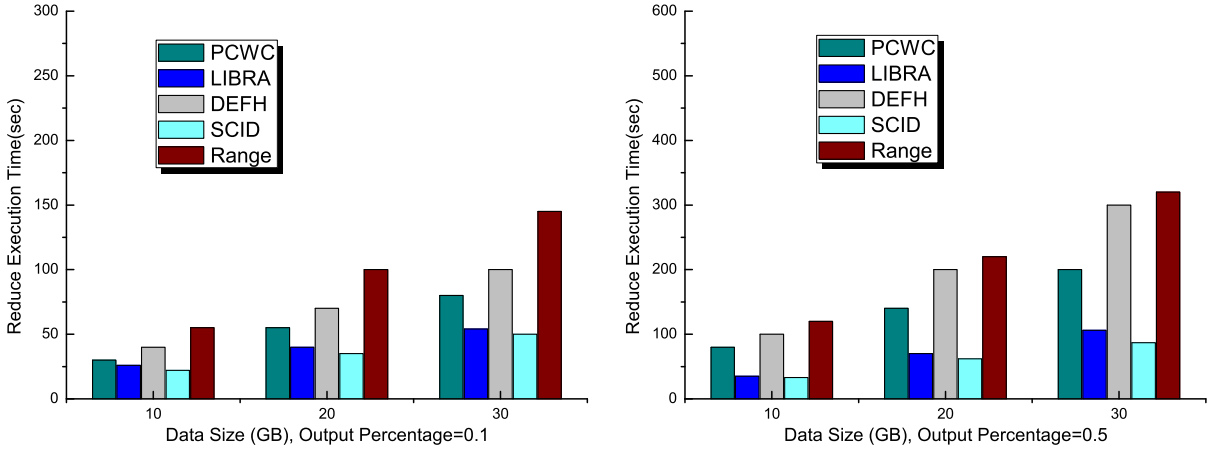
As shown in Fig. 11(b), compared with the results in Fig. 11(a), when the output percentage equals to 50%, for algorithm SCID, its difference to Range and SCID grows to the largest. That is due to the advantages of improving the load balance among reduce tasks for the higher skewed data.

Fig. 12 illustrates a group of experiments to evaluate the effect of balancing the reduce tasks workloads under the *Text Search* benchmark with various data sizes. As shown in Fig. 12(a), with a smaller query percentage, there is a lower growth rate of *FoS* based on the algorithms LIBRA and SCID when the input data sizes range from 10 to 30 GB, however, SCID is always better than LIBRA. Compared with the larger output percentages in Fig. 12(b), Fig. 12(a) can demonstrate that SCID has a significant load balancing advantage with higher data skew more clearly.

6.2.4. *Word count benchmark testing*

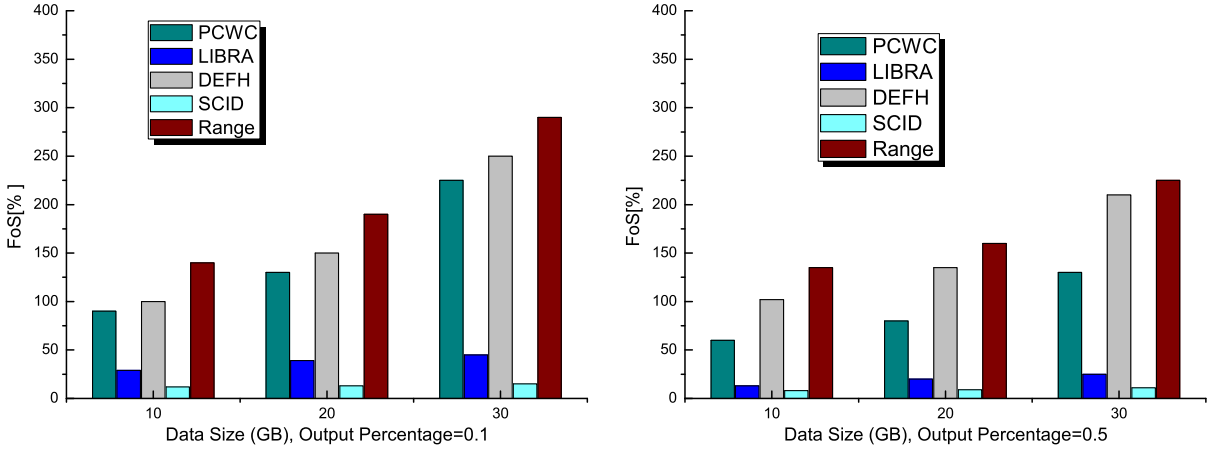
Fig. 13 shows the time performance and load balance capacities of algorithm SCID under *Word Count* benchmark. *FoS* values

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027



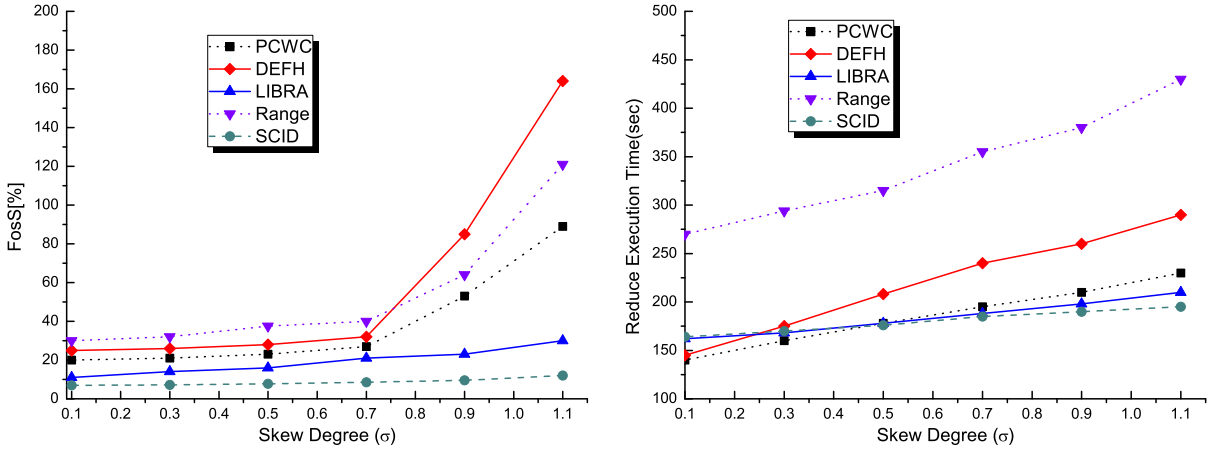
(a) *Output percentage* = 0.1. (b) *Output percentage* = 0.5.

Fig. 11. Reduce execution time vs. data size for *Text Search*.



(a) *Output percentage* = 0.1. (b) *Output percentage* = 0.5.

Fig. 12. Load balancing vs. data size for *Text Search*.

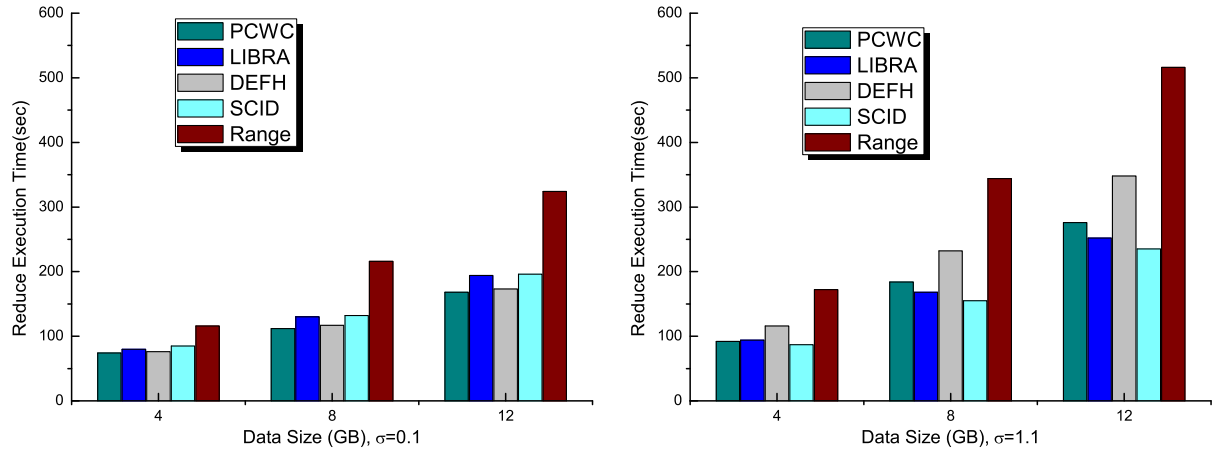


(a) Load balancing. (b) Execution time.

Fig. 13. Performance evaluation for *Word Count* benchmark as the data skew increases.

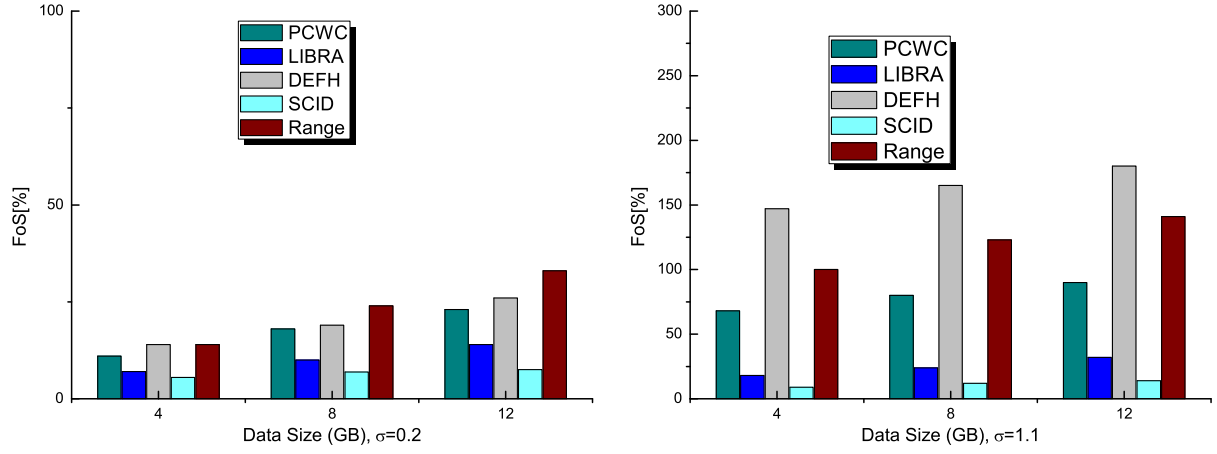
of buckets workloads and the finish time of reduce tasks are measured when the skew degree of the input data range from 0.1 to 1.1. Fig. 13(a) compares the capacities of load balancing for different algorithms. From Fig. 13(b), we can learn that SCID has less obvious performance advantage when the skew degree is low. The reason is similar to the explanation of Fig. 7(b) in the experiment under the benchmark sort: there are some additional calculations to determine the dispatching strategies before sending

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027



(a) $\sigma = 0.1$. (b) $\sigma = 1.1$.

Fig. 14. Reduce execution time vs. data size for *Word Count*.



(a) $\sigma = 0.2$. (b) $\sigma = 1.1$.

Fig. 15. Load balancing vs. data size *Word Count*.

the key/value tuples to the relevant buckets, which will bring the redundant overheads.

If the load balance mechanism cannot bring a larger performance improvement for the evenly distributed data, SCID is not better than other algorithms on time performance. But as the skew degree of the input data increases, the effect of load balance becomes more and more obvious. In more details, as shown in Fig. 13(b), the value of *FoS* increases rapidly when the degree of skew exceeds 0.7 for all experimental algorithms.

Fig. 14 represents the variation of reduce execution time with different skew degrees ($\sigma = 0.1$ and $\sigma = 1.1$), and the experimental results are measured with different data size: 4, 8, and 12 GB.

As shown in Fig. 14(a), when the data set has a lower skew degree, PCWC is the most time-efficient, and LIBRA as well as SCID performs even worse than DEFH due to the extra transmission of intermediate data. But in Fig. 14(b), as the above analyzed reasons, SCID can achieve a relatively better performance with greater skew for the input data. This can be verified from the measured results: the execution time of SCID grows more slowly than other algorithms as the data size increases. In these two experiments, Range performs worst among all algorithms, and the reason is similar to the analysis in sort benchmark experiment.

The squares in Fig. 15 confirms the differences of the balancing capacities for the compared algorithms under *Word Count* benchmark. Be similar to the analysis in *sort* benchmark, it is obvious that SCID has a significant advantage for load balance when the skew degree is higher.

6.2.5. Evaluation of whole execution time

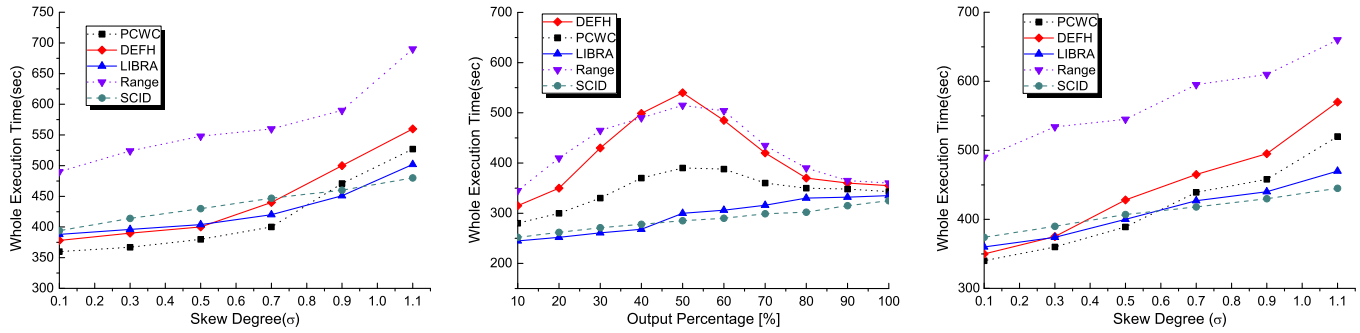
The implement of overall process of SCID algorithm includes two regular Spark jobs: the one is the sampling job, which is responsible to work out the key distributions for current input data. The main process is as the descriptions of the steps in Algorithms 1–3. The other is the original job, which is responsible for the specific business logics. Some improvements as the additional modules in the second job are as Algorithm 5. This may lead to some performance loss for the reduce tasks, and the reasons are explained and analyzed in Section 6.2.2.

Eq. (23) formalizes the components of the whole execution time: wet. In this equation, $(Sample)_{job1}$ is the whole execution time of the first sampling job, $(Map)_{job2}$ is the time cost of the map processing of the original job, $(Dispatching)_{job2}$ represents the time cost to determine the dispatching of the intermediate data output from map tasks, and $(Reduce)_{job2}$ represents the remaining time of this job.

$$wet = (Sample)_{job1} + (Map)_{job2} + (Dispatching)_{job2} + (Reduce)_{job2} \quad (23)$$

where $(Dispatching)_{job2} + (Reduce)_{job2}$ represents the reduce execution time in previous experimental results. Fig. 16 shows the whole execution time under these three benchmarks: *Sort*, *TextSearch*, and *Word Count*. Considering the four compared algorithms: PCWC, DEFH, LIBRA, and Range, they all have corresponding data sampling in their practical processes. For the

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027



(a) *Sort*. (b) *Text search*. (c) *Word count*.

Fig. 16. Whole execution time vs. data skew degrees for three Benchmarks.

same benchmarks and the same input data, the time costs in map phase are consistent in the rough. From Eq. (23), we can know that the differences among the whole time cost of these five algorithms mainly appear in the sampling and reduce phases. In all implementations of these algorithms, to keep things fair, the sample rates are all set as 20%. For SCID, the time cost $(Sample)_{job1}$ should be the duration to calculate out the distribution matrix of intermediate keys using 20% of the input data.

Moreover, the ratio of execution time between map and reduce phases is not only about data skew degree, but also about the repetition rate of intermediate keys. In extreme cases, for the input data with high repetition, because many works would be completed by the operations *combineByKey* in the map phases, the time cost in map phases would make up a larger proportion in the whole execution time. In our experiments, because the keys in the input data are disperse with a lower repetition, it makes the reduce time have considerable proportion in the whole time.

From Fig. 16(a), the slopes of the curves are roughly consistent with Fig. 7(b), which verifies the correctness of the previous analysis. For our fine grained data distribution polices, the advantages of SCID will be highlighted only when facing to higher skewed data. The higher skewed the data, the more obvious the effect. Moreover, it will raise the task waiting queues from the unbalanced workloads only if the data size exceeds a certain level. Hence, the advantages of algorithm SCID will also be relatively obvious for massive input data. For similar reasons, the slopes of Fig. 16(b) and (c) are also consistent with Fig. 10(b) and Fig. 13(b).

7. Conclusion

For the output of the map tasks, the default intermediate data dispatching mechanism in the Spark framework cannot achieve satisfied efficiency for the skewed data. Data skew mitigation is important for improving MapReduce performance. This paper first ascertains the distribution of the inner keys through a proposed sampling algorithm based on reservoir model. Because data sampling is always an additional work for the regular jobs, it will bring the extra running times and degrade the overall system performance inevitably. For this reason, based on the estimated frequencies of overall keys, this paper focuses on how to split and combine the output data from map tasks to the proper buckets rather than discuss when the sampling should start.

Experiments verify that the load balancing among the data containers for reduce tasks can be improved through data placement. And the system performance can also be improved for decreasing the waiting tasks in individual blocked queue. So the whole execution time of a job can be reduced to offset the time costs caused by the pre-running sampling in the real Spark implementation environment.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (Grant Nos. 61572176), National High-tech R&D Program of China (2015AA015303), and the Key Technology Research and Development Programs of Guangdong Province (2015B010108006).

References

[1] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[2] Q. Chen, C. Liu, Z. Xiao, Improving mapreduce performance using smart speculative execution strategy, IEEE Trans. Cloud Comput. 63 (4) (2014) 954–967.
[3] Thilina Gunarathne, Bingjing Zhang, Tak-Lon Wu, Judy Qiu, Scalable parallel computing on clouds using Twister4Azure iterative MapReduce, Future Gener. Comput. Syst. 29 (4) (2013) 1035–1048.
[4] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Symp. Oper. Syst. Des. Implement. (2004) 137–150.
[5] Andrei Sfrent, Florin Pop, Asymptotic scheduling for many task computing in Big Data platforms, Inform. Sci. 319 (2015) 71–91.
[6] Spark, 2015-11-30. [EB/OL] http://spark.apache.org/.
[7] B. Heintz, A. Chandra, R. Sitaraman, J. Weissman, End–to–end optimization for geodistributed mapreduce, IEEE Transactions on Cloud Computing (1) (2014) 1. http://dx.doi.org/10.1109/TCC.2014.2355225.
[8] J. Polo, Y. Becerra, D. Carrera, M. Steinder, I. Whalley, J. Torres, E. Ayguade, Deadline-based mapreduce workload management, IEEE Trans. Netw. Serv. Manag. 10 (2) (2013) 231–244.
[9] S.R. Ramakrishnan, G. Swart, A. Urmanov, Balancing reducer skew in mapreduce workloads using progressive sampling, in: Proceedings of the Third ACM Symposium on Cloud Computing. http://dx.doi.org/10.1145/2391229.2391245.
[10] Y. Le, J. Liu, F. Ergun, D. Wang, Online load balancing for mapreduce with skewed data input, in: INFOCOM 2014 Proceedings, IEEE, 2014, pp. 2004–2012.
[11] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, S. Wu, Handling partitioning skew in mapreduce using leen, Peer Peer Netw. Appl. 6 (4) (2013) 409–424.
[12] Alexandru Iosup, Hui Li, Mathieu Jan, Shanny Anoep, Catalin Dumitrescu, Lex Wolters, Dick H.J. Epema, The grid workloads archive, Future Gener. Comput. Syst. 24 (7) (2008) 672–686.
[13] Y. Xu, P. Zou, W. Qu, Z. Li, K. Li, X. Cui, Sampling-based partitioning in mapreduce for skewed data, in: ChinaGrid Annual Conference (ChinaGrid), 2012, pp. 1–8.
[14] Q. Chen, J. Yao, Z. Xiao, Libra: Lightweight data skew mitigation in mapreduce, Parallel and Distributed Systems, IEEE Transactions on Cloud Computing 99 (2014) 1–14.
[15] Fan Zhang, Junwei Cao, Samee U. Khand, Keqin Lie, Kai Hwang, A task-level adaptive MapReduce framework for real-time streaming data in healthcare applications, Future Gener. Comput. Syst. 43–44 (2015) 149–160.
[16] Word Count, 2015-11-30. [EB/OL] http://spark.apache.org/examples.html#WordCount.
[17] Y. Fan, W. Wu, Y. Xu, H. Chen, Improving mapreduce performance by balancing skewed loads, Communications 11 (8) (2014) 85–108. China.
[18] Radu Prodan, Specification-correct and scalable coordination of Grid applications, Future Gener. Comput. Syst. 23 (4) (2007) 587–605.
[19] Sort, 2015-11-30. [EB/OL] http://sortbenchmark.org/.
[20] Text Search, 2015-11-30. [EB/OL] http://spark.apache.org/examples.html#TextSearch.
[21] F.N. Afrati, J.D. Ullman, Optimizing joins in a map-reduce environment, in: Proceedings of the 13th International Conference on Extending Database Technology, ACM, 2010, pp. 99–110.
[22] M.A.H. Hassan, M. Bamha, F. Loulergue, Handling data-skew effects in join operations using mapreduce, Procedia Comput. Sci. 29 (2014) 145–158.

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027

[23] S. Blanas, J.M. Patel, V. Ercegovac, J. Rao, E.J. Shekita, Y. Tian, A comparison of join algorithms for log processing in mapreduce, in: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data, ACM, 2010, pp. 975–986.
[24] D.G. Mestre, C.E.S. Pire, Improving load balancing for mapreduce-based entity matching, in: 2013 IEEE Symposium on Computers and Communications. ISCC, IEEE, 2013, pp. 618–624.
[25] V.S. Martha, W. Zhao, X. Xu, h-mapreduce: a framework for workload balancing in mapreduce, in: Advanced Information Networking and Applications, AINA, 2013 IEEE 27th International Conference on, IEEE, 2013, pp. 637–644.
[26] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, L. Qi, Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud, in: 2010 IEEE Second International Conferenceon on Cloud Computing Technology and Science, CloudCom, IEEE, 2010, pp. 17–24.
[27] B. Gufler, N. Augsten, A. Reiser, A. Kemper, Load balancing in mapreduce based on scalable cardinality estimates, in: 2012 IEEE 28th International Conference on Data Engineering, ICDE, IEEE, 2012, pp. 522–533.
[28] B. Gufler, N. Augsten, A. Reiser, A. Kemper, T.U.M. Mnchen, Handling data skew in mapreduce, in: Proceedings of the 1st International Conference on Cloud Computing and Services Science, CLOSER 2011, Noordwijkerhout, Netherlands, 7–9 May, 2011, pp. 1–6.
[29] J. Lin, et al. The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce, in: 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2012, pp. 2000–2009.
[30] L.A. Adamic, B.A. Huberman, Zipfs law and the internet, Glottometrics 3 (1) (2002) 143–150.
[31] Daewoo Lee, Jin-Soo Kimb, Seungryoul Maenga, Large-scale incremental processing with MapReduce, Future Gener. Comput. Syst. 36 (2014) 66–79.
[32] RangePartitioner, [EB/OL] http://spark.apache.org/docs/1.3.0/api/java/org/apache/spark/RangePartitioner.html.
[33] Wikipedia, 2015-11-30. [EB/OL] http://en.wikipedia.org/wiki/Archive.

Zhuo Tang received the Ph.D. in Computer Science from 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 he is the sub-dean of the department of Computing Science. His major interests are in distributed computing system, cloud computing, and the parallel process for big data, including the distributed machine learning, security model, parallel algorithms, and resources scheduling and management in these areas. He is a member of ACM and CCF.

Xiangshen Zhang is working towards the master degree at the College of Information Science and Engineering, Hunan University, China. His research interests include the parallel computing, the improvement and optimization of task scheduling module in Hadoop and Spark platforms.

Kenli Li received the Ph.D. in Computer Science from Huazhong University of Science and Technology, China, in 2003. Now he is a Professor of Computer Science and Technology at Hunan University, Associate Director of National Supercomputing Center in Changsha. His major research includes parallel computing, grid and cloud computing, and DNA computer. He has published over 260 journal articles, book chapters, and refereed conference papers. And he is an outstanding member of CCF and a member of IEEE.

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 over 490 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 IEEE Transactions on Parallel and Distributed Systems, IEEE Transactions on Computers, IEEE Transactions on Cloud Computing, Distributed Computing. He is an IEEE Fellow.

Please cite this article in press as: Z. Tang, et al., An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems (2016), http://dx.doi.org/10.1016/j.future.2016.06.027