Ibrahim *et al. Journal of Cloud Computing: Advances, Systems and Applications* (2020) 9:8
https://doi.org/10.1186/s13677-019-0139-6

Journal of Cloud Computing:
Advances, Systems and Applications

RESEARCH Open Access

Improvement of job completion time in data-intensive cloud computing applications

Ibrahim Adel Ibrahim[1,2*] and Mostafa Bassiouni[3]

Abstract

Task stragglers in MapReduce jobs dramatically impede job execution of data-intensive computing in cloud data centers. This impedance is due to the uneven distribution of input data, heterogeneous data nodes, resource contention situations, and network configurations. Data skew of intermediate data in MapReduce job causes delay failures due to the violation of job completion time. Data-intensive computing frameworks, such as MapReduce or Hadoop YARN, employ HashPartitioner. This partitioner may cause intermediate data skew, which results in straggler reducers. In this paper, we strive to make Hadoop YARN more efficient in cloud environments. We present, a new partitioning scheme, called balanced data clusters partitioner (BDCP), to handle straggler Reduce tasks based on sampling of input data and feedback information about the current processing task. Our extensive experimental results show that BDCP can outperform the default Hadoop HashPartitioner and Range partitioner. BDCP can assist in straggler mitigation during reduce phase and minimize the job completion time in MapReduce jobs within data-intensive cloud computing.

Keywords: Cloud computing, MapReduce, Data-intensive computing, Parallel and distributed processing, Straggler reduce task, Sampling

Introduction

The rapid growth of information and data in the age of data explosion in industry and research poses tremendous opportunities, as well as tremendous computational challenges. To manage the immense volumes of data, users have needed new systems to scale out computations to multiple nodes. Modern cloud data centers are composed of thousands of servers to support the increasing demand on cloud computing.

Due to the large scale of the data-intensive jobs, the only feasible way to solve them while fulfilling Quality of Service (QoS) requests is to partition them into small tasks which can be processed in parallel across many computing nodes [1]. MapReduce, designed by Google, has been widely used as the most popular distributed programming model for parallel processing of massive datasets (usually greater than 1 TB) in cloud environments. It divides a large computation into small tasks and assigns them to multiple computational cluster of nodes running in parallel. MapReduce is unique in reliability, and scalability to large clusters of inexpensive commodity computers. It divides the job into multiple tasks, and handles tasks execution and the complexity of fault tolerance in a distributed manner [2].

MapReduce consists of two main phases: map, and reduce. Tasks are distributed to cluster of processing nodes during map and reduce phases. During map phase, chunks of the huge data sets are processed concurrently on individual computers in the cluster. Reduce phase has 3 steps: shuffle, sort, and reduce. Shuffle starts when the data is collected by the reducer from each mapper. Shuffle may start when mappers have generated enough amount of data. On the other hand, sort and reduce can start once all the mappers have finished map phase and the resulted intermediate data have been shuffled to the reducers. Reduce phase combines the intermediate data from map phase and derives the final output.

Big data tools like Hadoop and Apache Spark provide productive high-level programming interface for large scale data processing and analytics. Hadoop uses MapReduce as programming paradigm. It has been used for

*Correspondence: efahad@knights.ucf.edu
[1]Department of Computer Engineering, University of Central Florida, Florida, USA
[2]Department of Computer Engineering, University of Technology, Baghdad, Iraq
Full list of author information is available at the end of the article

parallel processing of large-scale data on a large cluster of commodity machines to handle data-intensive applications. The next generation of Hadoop, namely Hadoop YARN, is accommodated to various programming frameworks and capable of handling many kinds of workload such as interactive analysis, and stream processing. In a MapReduce cluster, the job is submitted, then it is divided into multiple map tasks.

Map tasks extract (Key, Value) pairs from the input data chunks. All (Key, Value) pairs sharing the same key form data cluster. The total number of (Key, Value) pairs in the data cluster is the data cluster size. Map outputs generate the intermediate data. The intermediate data are divided according to a user defined partitioner before being sent to the reducers [3]. Thus, the mapper groups the data clusters into partitions. The partition is a set of data clusters assigned to the same reducer. Therefore, the number of partitions in each mapper equals the number of reducers. Every partition from each mapper is sent to the corresponding reducer, as shown in Fig. 1.

The default partitioner of Hadoop is HashPartitioner. Since all map tasks use the same partitioner, all similar keys are dispatched to the same partition. Every partition consists of many data clusters. The number of data clusters is equal to the number of distinct keys in the input data. One reducer processes one data partition.

Ideally, the resulted intermediate pairs of keys and values consist of different keys that are approximately equal in their values. In this ideal case, the reducers process same amount of data because all reducers process same number of data clusters. However in real applications, the reducers vary in their assigned intermediate data because

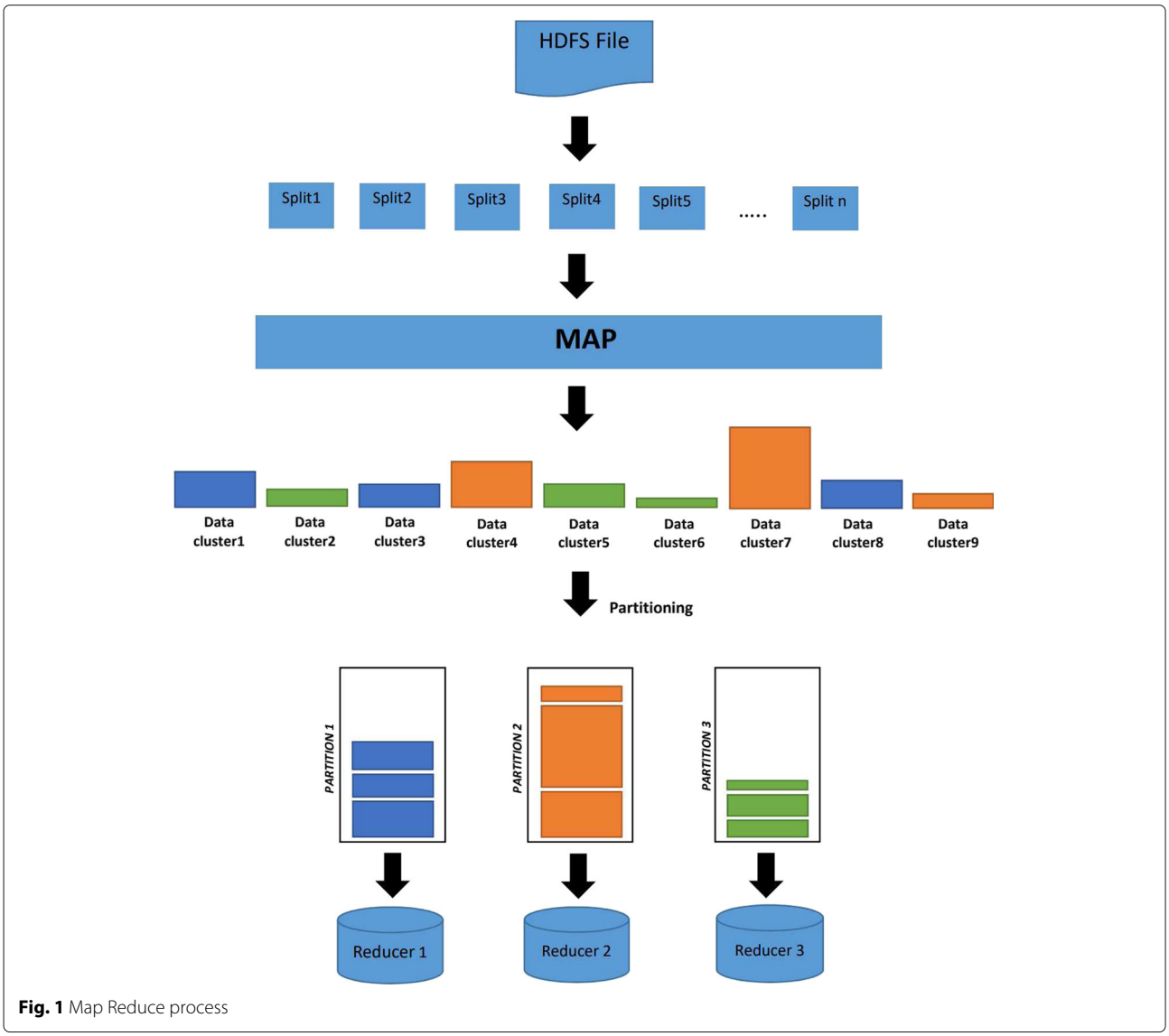

Fig. 1 Map Reduce process

the data clusters vary in their sizes, and the reducers vary in the number of data clusters they process. Moreover, one reducer may have been assigned too much data to process as compared to the other reducers for the same job. Consequently, all other reducers complete their reduce tasks while heavy reducer becomes straggler reducer.

The straggler reducers degrade the performance of MapReduce applications because the result of the reduce phase is computed after receiving the results of all reduce tasks including straggler reduce task. In cloud computing platform, the reduce task that receives extremely large data becomes straggler, eventually delays the overall job completion time. Straggler problem is very common in reduce tasks in data-intensive MapReduce jobs because of three major reasons:

- The data skew resulted from the partitioner: In case of data skew, the resulted data load on a reducer is much higher than the data load on the other reducers for the same job. In Hadoop, data skew happens because of the keys dispatching is based on the hashing algorithm. The size of partition depends on the number of relevant (Key, Value) pairs. Data skew is one of the main performance bottlenecks in MapReduce environment.
- The variations in computing capabilities of reducers: In heterogeneous hadoop cluster the data nodes may vary in data processing speed due to the diversity of their computing capabilities [4]. Even if there is no data skew, the variations in computing capabilities of data nodes that perform the reduce tasks lead to the case in which the slow node becomes straggler.
- The network congestion: It is resulted from the huge amount of data transferred from mappers to the reducers during the shuffle phase. Since the reducer waits for all the data clusters to arrive in order to start reduce task, the delay in transferring the data needed by a reducer leads to straggler reduce task.

Partitioner controls the partitioning process of the keys generated in map phase. The key (or a subset of the key) is used to derive the partition, typically by a hash function. The total number of partitions is the number of reduce tasks of the job. During the reduce phase, large data partition may be assigned to one reducer while the other reducers receive small partitions, as shown in Fig. 1. Consequently, all other reducers complete their reduce tasks and wait for the large data reducer to process the large partition of data, which leads to delay of final result. For example, Hadoop 2.9.2 employs the following static hash function to partition the intermediate (Key, Value) pairs [5].

$$Hash\left[HashCode\,(Key)\ mod\ (numReducer)\right].$$

Unfortunately, hash function cannot solve the issue of skewed data. For reduce tasks, partitioning skew leads to shuffle skew, eventually some reducers receives extremely more data than others [6]. For example, Fig. 2 shows different amounts of input data that have been assigned to 12 reducers by using benchmark Word Count with 15 GB of text data [7].

Many straggler mitigation techniques have been developed in order to solve the issue of straggler. One of straggler mitigation techniques is the reallocation of straggler reduce task. In this technique, an alternative reducer is selected to run the reduce task. This process requires transferring all data of the reduce task to the new alternative reducer. In some cases, task reallocation leads to a higher overhead compared to the overhead produced by processing the task using the original slow node. This overhead resulted from the delay of transferring the data

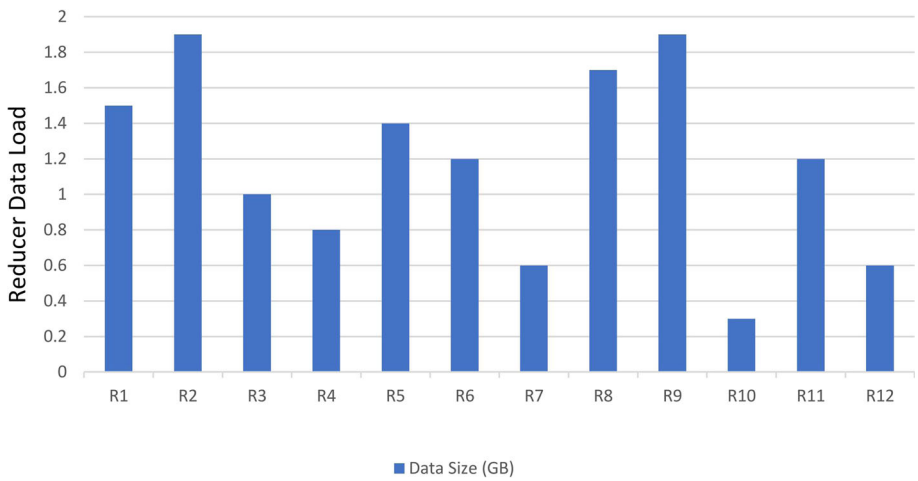

Fig. 2 Variation in reducers data load

over the network to the alternative node. However, the decision of transferring a reduce task to another reducer should be based on accurate calculations about the data transfer cost in order to minimize the completion time of the reduce phase.

The ideal way to avoid straggler reduce tasks issue is by trying to distribute the data clusters evenly as much as possible to the reducers. In order to distribute the intermediate data to reducers in an efficient way, the partitioning policy must be based on information about the (Key, Value) pairs resulted from each map task before the beginning of the shuffle process. The information must include the frequency of each key produced from every mapper in order to apply the partitioning policy that produce partitions that are similar in their sizes. Obtaining an ideal solution for this problem is unrealistic because of two reasons:

- In current Hadoop, the execution time of shuffling the mappers outputs to the reducers is overlapped with the map tasks execution time. Reduce phase is activated when specific percentage of map tasks have been completed, (5% by default in Hadoop 2.9.2). Overlapping the execution of map tasks and reduce tasks is handled in order to avoid network congestion, and fully utilize the resource. Consequently, minimizing the job completion time.
- An accurate information about the intermediate data can be obtained only when all map tasks have been finished. However, it is meaningless to obtain the (Key, Value) distribution after processing all input data in map phase, because the cost of pre-scanning the whole data is hard to be accepted when the amount of data is very large.

Furthermore, when the amount of input data is very large, the job completion time in case of waiting for all map task to finish then starting the reduce phase is higher than the job completion time when the current default hash policy has been used [8].

To counter this problem, we propose a strategy to mitigate straggler reducer in MapReduce job on massive datasets, namely, Balanced Data Clusters Partitioner(BDCP). BDCP makes an estimation of the whole intermediate data, by taking sufficient sample of the input data, applies the MapReduce job on the sampled data using the mappers and reducers assigned to the job, then starts the actual MapReduce job on the actual input data. It can balance the load on reducers based on the estimation of the sizes of intermediate data and the current data processing capacity of the reducers. The extensive experiments show that this policy produces shorter job completion time than the default partitioner of Hadoop 2.9.2 and Range partitioner [9].

The rest of this paper is organized as follows. "Background and related work" section introduces the background and related work. "Design overview" section describes the proposed policy. "Evaluation" section presents the evaluation results. We conclude in "Conclusions" section.

Background and related work

Due to its importance in data-intensive cloud computing, the subject of straggler identification and tolerance has received considerable amount of research attention. The mechanism of speculative execution is used in MapReduce to address the straggler problem. Speculative execution performs backup execution of the remaining running tasks when the parallel processing is close to completion. There are numerous speculation-based techniques for straggler-mitigation. SkewTune [10] re-partitions the data of stragglers to move it to idle slots resulted after completing the processing of short tasks. However, moving re-partitioned data to idle nodes may lead to nodes communication overload, which could negatively impact the computing performance.

Restarting reduce tasks on another node requires transferring the whole amount of data over the network. Recently, many algorithm and models about reduce tasks scheduling have been proposed. Hassan et al. [11] proposed a MRFA-Join algorithm , it 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.

Data skew and load balancing problem is one of the main reasons of straggler reducers. In order to achieve balanced load, many researchers have focused on designing a new parallel programming model based on MapReduce [12–14]. LIBRA has been proposed in [5]. It is a lightweight strategy to resolve the data skew problem, it applies a sampling technique to produce an accurate estimation of the distribution of the intermediate data. It samples part of the intermediate data during the map phase. LIBRA supports large cluster and it works for heterogeneous environments, but the partitioning does not consider the current processing load of the reducers.

Yu et al. [15], use sampling MapReduce job to gather the distribution of keys' frequencies, make estimation of the overall distribution, then partition scheme is generated in advance. Two partition schemes have been proposed based on sampling results: cluster combination and cluster partition combination. The idea of cluster combination is that the biggest data cluster is assigned to the reducer with the smallest load in order to achieve the load balancing of all reducers. The cluster partition combination is used when the skew in intermediate data is very high. In this case the large cluster is divided into equal pieces, and then, every piece is assigned to a reducer. This method breaks the rule that each partition should be processed by

a single reducer. An additional reduce phase is configured to merge the results generated from multiple reducers.

Tang et al. [16], have proposed splitting and combination algorithm for skew intermediate data blocks (SCID). The sampling is used to predict the distribution of the keys in intermediate data. In SCID, the data clusters are sorted, and for each map task the output filled into buckets. A data cluster must be split once it exceeds the residual volume of the current bucket. After filling this bucket, the remainder cluster will be started the next iteration. The main idea is that each reduce task gets its share of intermediate data from particular bucket of map task. SCID focuses on how to split and combine the output data from map tasks to the proper buckets rather than decide when the sampling should start. This method split a big partition to be processed by more than one reducer.

In our previous work [17], we proposed Progress and Feedback based Speculative Execution Algorithm (PFSE). It is a new Straggler identification scheme to identify the straggler tasks in MapReduce jobs based on the feedback information received from completed tasks, and the progress of the non-completed processing task. (PFSE) focuses on map phase and sort part of reduce phase only. Dolly [18], provides a speculative execution at job level which clones small jobs with straggler tasks. Dolly employs a technique called "delay assignment" to avoid contention of intermediate data. It launches multiple clones of every task, the output that completes first is used while the other clones are neglected. But duplication of the entire job most probably increases the resource usage and I/O contention on data.

Zaharia et al. [19] suggested Longest Approximate Time to End (LATE), a modified version of speculative execution. It allows Hadoop to speculatively execute the task that expected to be delayed. Instead of considering the progress made by a task, LATE computes the estimated remaining time to complete the task. LATE depends on HDFS for replica placement, this restriction minimizes the number of tasks that involved in the speculative execution. LATE is designed to enhance Hadoop performance in both homogeneous and heterogeneous environments. The experimental results indicate that LATE can improve the job completion time of Hadoop by a factor of two.

Xie et al. [20] are partitioning the data according to cluster nodes capabilities. Slow nodes receive less data load than faster nodes. This static profiling does not consider other loads that may begin share the node with the current MapReduce job. The drawback of this work is when the predicted node to receive more load fail, the job takes longer time than if the slower node has been used.

Lin et al. [21] proposed Self-Learning MapReduce scheduler (SLM) to improve the speculative algorithm in a multi-job cloud platform. SLM uses feedback information collected from some recently completed tasks of the same job to calculate the phase weights. However, SLM gets better accuracy of estimation with the progress of time because it needs to determine a specific number of finished tasks to use it for learning process.

All these approaches lack the ability to identify the performance bottleneck of the straggler tasks. Speculative copies of tasks perform better but it causes resources overload, and it is not always successful solution. Load imbalance among cluster nodes is a major reason for the occurrence of stragglers in parallel processing. In our previous work [22] we address the load balancing issue from the perspective of balancing replicas assignment across all cluster nodes of Hadoop, and propose a replica placement policy that run an algorithm to distribute the replicas across all cluster nodes based on their data load.

In [23], we address the load balance issue of Hadoop from two different perspectives: task assignment and replica placement mechanism. We present two improved replica placement policies for Hadoop Partition Replica Placement Policy (PRPP) [24], and Slot Replica Placement Policy (SRPP) [25].

Design overview

An algorithm named Balanced Data Clusters Partitioner BDCP has been developed to improve MapReduce performance. This algorithm reduces the MapReduce job execution time by addressing the problem of straggler reducers caused from skewed data, network overhead, and slow reducers, by the following contributions:

1. Minimizing the effect of intermediate data skew.
2. Preventing the reducers skew by balancing the data load on the reducers.
3. Minimizing the amount of data transfer during shuffle phase over the network from mappers to the reducers.

The main steps of this algorithm are summarized as following:

1. Implementing the MapReduce job on a small sample from each split of the input data.
2. Calculating the estimated frequency of every key .
3. Collecting feedback information about the computing capabilities of reducers.
4. Building a new partitioning policy of the intermediate data for heterogeneous and homogeneous reducers nodes.

BDCP policy starts the sampling phase before the actual execution of MapReduce job. In sampling phase, a sample from the data input of each map task is taken. The sampling process used must ensure an accurate data representation of the original data in the sample data. The original MapReduce job is applied on the sampled data

by using the same mappers and reducers reserved for this job. Once the information about the intermediate data and the reducers processing capabilities are received by the partitioning algorithm, the partitioning policy is created, and the MapReduce job starts to be implemented on the original data.

The efficiency of the algorithm depends on the size of data sample, the accuracy of sampling method used, and the accuracy of feedback information about the reducer nodes. The bigger the size of data sample used, the more accuracy of the estimation of the key frequency for the original data.

However, the bigger size of data sample used, the longer time it takes to complete the sampling phase, consequently longer job completion time. Sample size %10 of the original input data is used in the experiments in policy evaluation. The main processing steps of MapReduce job with BDCP partitioning policy is shown in Fig. 3. At the beginning, the reservoir sampling takes a sample of input data from each split using efficient sampling method as discussed in next section, then the regular MapReduce job is implemented on the sampled data. The intermediate data are partitioned using the default HashPartitioner. The reducers that are assigned to the original job is used to reduce the intermediate data in sampling phase. The last part of sampling phase is that the partitioning policy part of BDCP receives the results. However, the NameNode receives the information about speed of data processing of the reducers through the heartbeats. Thus, during the sampling phase and before the actual execution of MapReduce job, an accurate information about the data processing tare of the reducers are available in the NameNode to be used in the distribution policy. The distribution policy uses a modified knapsack problem algorithm. It assumes the reducers are the buckets with one size or different sizes, and data clusters are the items that need to be placed

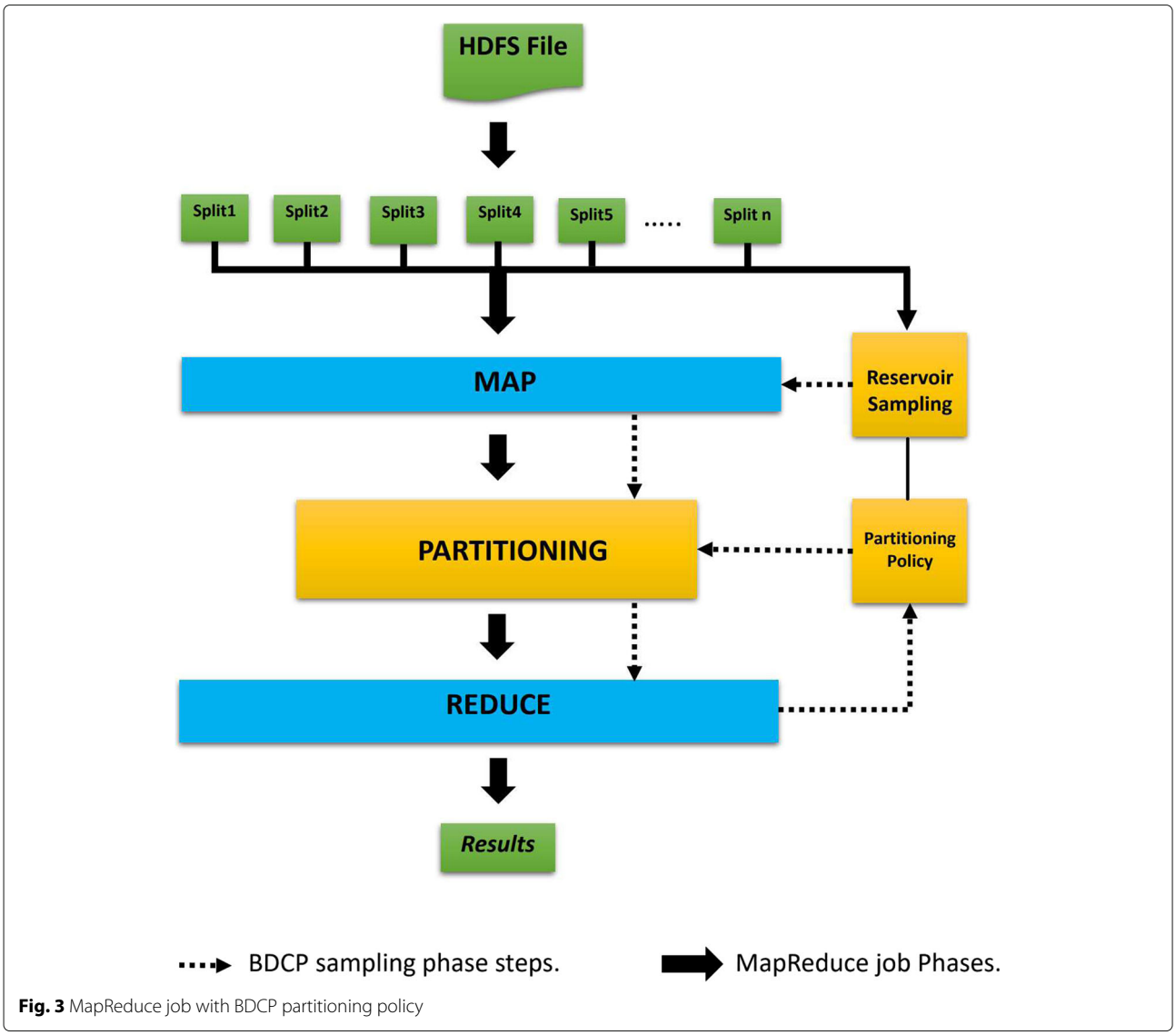

Fig. 3 MapReduce job with BDCP partitioning policy

inside the buckets. The size of the reducer is based on the data consumption rate received by NameNode.

Sampling

Sampling is the selection of a subset (representative sample) from a target population then collecting data from that sample in order to estimate characteristics of the whole population. It is an efficient tool to reduce the amount of input data and dealing with a sample as a representative for the original data. Even though there are many sampling techniques, the type and features of data determine the sampling method that makes best representation of the original data. For example, if the data set is already sorted and a sampling method needed to find the distribution of this data set, then the best sampling technique is the interval sampling method. When a general information about the density distribution of numbers, and the probability about data distribution are known, then a probability sample may be used.

In probability sample every unit in the population has a chance of being selected in the sample, and this probability can be accurately determined. When the data set is entirely unknown, it is best to apply simple random sampling. In a simple random sample (SRS) of a given size, all such subsets of the frame are given an equal probability. Each element of the frame thus has an equal probability of selection. Since the Hadoop MapReduce processes different kind of data with different features, it is best to use simple random sample. In practical applications of Hadoop MapReduce, the amount of data is very large, therefore BDCP use the sampling phase. The number of (Key, Value) pairs for each key in sample is approximately the proportion of (Key, Value) pairs in the original input data. The number of (Key, Value) pairs sharing the same key appear in the sample can be scaled up by dividing it by the sampling ratio to produce the estimated frequency of this Key in the original data, as in Eq. (1).

$$size(k) = \frac{size\left(k^{'}\right)}{S\_Ratio} \tag{1}$$

S_Ratio is the sampling ratio, k is the original key, and $k^{'}$ is the key that appears in the sample.

Reservoir sampling

The reservoir sampling takes k elements from the population. It saves k preceding elements first, then randomly replace original selected elements in the reservoir with a new element that is selected from outside the reservoir. The final sample data of size k is generated after finishing the scanning of all the original input data, as shown in Algorithm 1. Assume S is the data population and the required sample size is k. The algorithm creates a "reservoir" array of size k, and directly place first k items of S in it. It then iterates through the remaining elements of S, beginning from the $(k+1)^{th}$ element until S is exhausted. At the i^{th} element of the iterations, the algorithm generates a random number j between 1 and i. If j is less than or equal to k, j^{th}element of the reservoir array is replaced with the i^{th} element of S.

In effect, for all i, the i^{th} element of **S** is chosen to be included in the reservoir with probability k/i. Similarly, at each iteration the j^{th} element of the reservoir array is chosen to be replaced with probability $\left(\frac{1}{k}\right) \times \left(\frac{k}{i}\right) = \left(\frac{1}{i}\right)$.

We analyze the time complexity of Algorithm 1 as follows: Line 2 takes O(1) time. The time of the loop in lines 3–12 depends on N, where Data blocks consist of N records to be assigned. Therefore, the time complexity of this loop is O(N). If statement in lines 4–12 take O(1) time. So the total time complexity of sampling algorithm is O (N) .

Theorem: When the Reservoir Sampling algorithm has finished sampling process on a data set, each item in the data set has gotten equal probability of being chosen for the reservoir.

Proof: Let's assume that a sample of size k representing data set of size S. We are required to prove that each item in S has gotten equal probability of being chosen for reservoir. As shown in Fig. 4, assume the algorithm is in the $(i-1)^{th}$ round, x is the element of the $(i-1)^{th}$ round, it either has been selected as a sample in the reservoir or has been skipped. The probability of x being selected in the reservoir array after completing round $(i-1)$ is $\left(\frac{k}{i-1}\right)$.

Since the probability of the j^{th} element of the reservoir array is chosen to be replaced in the i^{th} round is $(\frac{1}{i})$, the probability that x survives inside the reservoir in the i^{th} round is $\left(\frac{i-1}{i}\right)$. Thus, the probability that x is in the

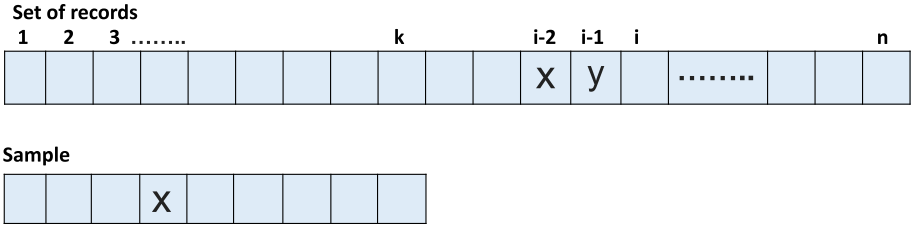

Fig. 4 Reservoir sampling

reservoir after the i^{th} round is the product of these two probabilities, i.e. the probability of being in the reservoir after the $(i-1)^{th}$ round, and probability of x staying inside reservoir in the i^{th} round: $\left(\frac{k}{i-1}\right) \times \left(\frac{i-1}{i}\right) = \frac{k}{i}$.

At the same time, the probability for the i^{th} element to be swapped in is also $\frac{k}{i}$. Hence, the result holds for i. Moreover, since the base case of $i-1 = k$ is true, the result is therefore true for all $i \geq k$ by induction.

In general, for $(k+1) < i \leq n$, probability of $S[i]$ being in the reservoir is: (Probability of selecting $S[i]$ to be in the reservoir in i^{th} round)$\times$(Probability of not removing $S[i]$ from the reservoir during the $(i+1)^{th}$ round)$\times$ (Probability of not removing $S[i]$ from the reservoir during the $(i+2)^{th}$ round)$\times ....\times$ (Probability of not removing $S[i]$ from the reservoir during the n^{th} round). It can be simplified as follows:

$$p(i) = \frac{k}{i} \times \frac{i}{i+1} \times \frac{i+1}{i+2} \times \frac{i+2}{i+3} ... \frac{n-2}{n-1} \times \frac{n-1}{n} = \frac{k}{n}$$

the result is true for all $(k+1) < i \leq n$.

Algorithm 1: Reservoir sampling.

Input: **Input**: $\mathbf{S_i}[1, 2, .., \mathbf{N}]$: Data block, it has **N** records.
Data blocks consist of **N** records to be assigned, which are numbered from 1 to (**N**)
Result: $R_i[1, 2, .., K]$: sample which has k records.

```
begin
    CB = 0; (Current block number)
    for i = 1 to n do
        if i ≤ k then
            R[i] = S[i];
        else
            j = random(1, i);
            if j ≤ k then
                R[j] = S[i];
            end
        end
    end
    Return R
end
```

Partitioning policy

Partitioning of the data clusters is the process of grouping data clusters into partitions. Every partition is assigned to a reducer to implement the reduce task. To increase the utilization of the resources in reduce phase, the reducers must be similar in their data load received during the shuffle phase. After finding an accurate sample that represents the original input data, partitioning policy of BDCP is decided. There are three important factors controlling the partitioning policy:

- Estimated (*key*, *value*) distribution.
- Reducers processing capacity.
- Network bandwidth.

Let's assume that the number of mappers and reducers are M, and R, respectively. The key and value sets are K, and V respectively. The intermediate data set, (*Key*, *Value*) pairs, generated during map phase is I. Data cluster is the set of all (*Key*, *Value*) pairs that sharing the same key. The number of data clusters in I is C. The data clusters, $Cluster_0$ through $Cluster_{(c-1)}$, need to be distributed into R reducers.

$k_i \in K$ is the key of $Cluster_i$ for $i = 1, 2, ..C$.
$Cluster_i = < k_i, v > \in I$ for $i = 1, 2, ..C$.

One partition, P_i, is a set of data clusters that assigned to $Reducer_i$. Therefore, the intermediate data I consists of partitions from P_1 through P_R: $I = (P_1, P_2, .., P_R)$. Let's assume a Hadoop Cluster consists of R reducers. C is the total number of data clusters appears in the sampling phase. The frequency of keys in data $cluster_i$ is key_i , where $(i = 0, 1, 2, ..., C-1)$. The total number of (*Key*, *Value*) pairs that need to be distributed to the reduce tasks is D. As in Eq. (2).

$$D = \sum_{i=1}^{C} key_i \tag{2}$$

Since data clusters vary in their sizes, and one data cluster must be assigned to one reducer, the sizes of partitions are not similar. Consequently, the distribution of data loads to the reduce tasks cannot be even. The mean size of the partitions, *Mean*, can be calculated as shown in Eq. (3).

$$Mean = \left(\frac{D}{R}\right) \tag{3}$$

The ideal case is when all data partitions are similar in their sizes, i.e. the partition size equals *Mean*. practically this case is unrealistic, but as much as the partitions sizes are close to *Mean* as much as the reduce phase is closer to the Ideal case. The suggested partitioning algorithm minimizes this variation, it tries as much as possible to minimize the standard deviation, s, of the number of (*Key*, *Value*) pairs on every reducer from *Mean*, as shown in Eq. (4).

$$\begin{aligned} \min_{T_{ij}} \ \mathbf{s} &= \sqrt{\frac{\sum_{i=1}^{R}\left(\left(\sum_{j=0}^{C} key_j T_{ij}\right) - Mean\right)^2}{n}} \\ \text{s.t.} \ \sum_{i=0}^{R} \mathbf{T}_{ij} &= 1, \text{for all } j = (1, .., C) \\ \mathbf{T}_{ij} &= 0 \ or \ 1, \text{for all } i = (1, .., R) \\ & \quad and \ j = (1, .., C) \end{aligned} \tag{4}$$

Where T is the main assignment table, it has R rows and C columns.

The distribution algorithm is shown in Algorithm 2. Data clusters assignment algorithm assigns C clusters to R partitions, one partition for one reducer. Every reducer pulls its data from its corresponding partition to achieve the reduce task.

Algorithm 2: Data clusters assignment algorithm-homogeneous

Input: **Input**: A collection of **R** reducers.
C data clusters to be distributed, which are numbered $Cluster_1$ through $Cluster_C$.
Number of keys in every data cluster i is $Key[\,i]$ where $(i = 1, 2, ..., \mathbf{C} - 1)$
Result: $T[\,1, 2, ..R]\,[\,1, 2, .., C]$

```
1  begin
2      for i = 1 to R do
3          Reducer_load [i] = 0;
4          T[ i] [ j] = 0; for all j = 1 to C
5      end
6      Sorted_key[ ] = sort_descending(Key[ ] );
7      Index_Sorted_key[ ]=index of Sorted_key[ ] in
       Key[ ] ;
8      for i = 1 to C do
9          j = Index_Sorted_key[ i];
10         if (keys of Cluster_j are the Map output of
           Mapper/Reducer node) && (this node
           produces more than half of data cluster j)
           then
11             z = Index(Mapper/Reducer node);
12         else
13             z = Minimum(Reducer_load[1,..,R]);
14         end
15         T[ z] [ j] = 1;
16         Reducer_load[ z] =
           Reducer_load[ z] +Key[ j];
17     end
18     Return  T [1,2,..R][1,2,..,C]
19 end
```

Lines 1-5 in Algorithm 2 clear the array called reducer_load, it stores the current load on the reducer. At the beginning of the distribution *reducer_load* for every reducer is 0. The algorithm creates $R \times C$ array to store the assignment table as shown in Fig. 5 . At the beginning of the algorithm, all values of main assignment table equal 0. During the process of the algorithm, if a cell inside this table has been assigned value of 1, then data cluster of corresponding column number is assigned to the reducer of corresponding row number in main assignment table.

It is important to sort the data clusters based on their size in descending order. The reason of this descending order is to start the assignment of the largest data clusters first then the smaller size. Leaving the data clusters with small sizes to the end of distribution make it easier to balance the load among reducers. In lines $6 - 7$, the function *Sort_descending* sorts the sizes of data clusters, *key*[ ], in descending order. The results are saved in *sorted_key*[ ] array and their original index in *Key*[ ] are stored in array named *Index_sorted_key*[ ]. For example: $Key = [\,22, 41, 11, 32]$, then $sorted\_key = [\,41, 32, 22, 11]$, and $Index\_sorted\_key = [\,2, 4, 1, 3]$. Line 9 takes the next data cluster from *sorted_key*[ ], and store its index on the original data clusters sequence.

Lines $10 - 11$ in Algorithm 2 is the part that minimize the data transfer over the network. A data node may be selected to execute both map and reduce tasks of a job, we call it Mapper/Reducer node. BDCP takes advantage of this opportunity to reduce the network overhead during the shuffle phase. The amount of data transferring over the network can be minimized by keeping the output of map task on a data node as an input of reduce task on the same node. Assume a data node X acts as Mapper/Reducer node for specific *MapReduce* job. In the proposed algorithm, if X produces more than half of the size of a data cluster during map task, the partitioning algorithm assigns this data cluster to X for reduce task.

Line 13 assigns the data cluster in the reducer with minimum load for the current iteration. The data load of the selected reducer is updated in line 16 by adding the size of the data cluster to its load. BDCP produces the Main Assignment Table (MAT), shown in Fig. 5. MAT

	Key_1	Key_2	Key_3	Key_4	.....	Key_C
$Reducer_1$	1	0	0	1	.....	0
$Reducer_2$	0	1	0	0	.....	0
$Reducer_3$	0	0	0	0	.....	1
⋮			⋮			
$Reducer_R$	0	0	1	0	.....	0

Fig. 5 Main assignment table

is the partitioner that used during the shuffle phase to map every key to its dedicated reducer. The final values of T_{ij} determine the reducer of every data cluster. During the assignment process, in every iteration line 15 assigns value of 1 to the selected row and column in the main assignment table.

We analyze the time complexity of Algorithm 2 as follows: Line 3 takes O(1) time, and line 4 takes O(1) time. The time of the loop in lines 2–5 depends on R, where R is the number of reducers. Therefore, the time complexity of this loop is O(R). The time of the loop in lines 8–17 depends on C, where C is the number of data clusters to be distributed. Therefore, the time complexity of this loop is O(C). So, the total time complexity of data assignment algorithm is O (N) + O(C).

The partitioning algorithm is different in heterogeneous cluster environments. Since the reducers are usually belonging to different hardware generations, reducers may differ in their processing capability. Name node receives, from every reducer, the reducer data processing rate through the heartbeats. Sampling phase is the perfect time for checking the current data processing speed of reducers before the starting of actual *MapReduce* job. *BDCP* assigns a processing ratio to the reducers and calculate the ratio of data every reducer should get to satisfy the balancing capacity. During the sampling phase, *BDCP* calculates, d_i, the amount of data processed between two successive heartbeats by reducer i. The processing ratio, p_i, of reducer i can be calculated by *BDCP* during the sampling phase as shown in Eq. 5.

$$p_i = \frac{d_i}{\sum_{j=1}^{R} d_j} \tag{5}$$

p_i is the approximate ratio that reducer i should get from the total intermediate data, where $(i = 1, 2, ...R)$. The size of data that should be assigned to reducer i, noted as sh_i, is calculated using Eq. 6

$$sh_i = p_i \times D \tag{6}$$

D is the total estimated data size of intermediate data. During data clusters distribution, every time a data cluster is assigned to partition, the share, sh_i of the reducer corresponding to this partition is decreased.

The partitioning algorithm in heterogeneous reducers is designed as a modified multiple knapsack problem, where every knapsack has a capacity, we called it balancing capacity. The load distributed evenly across knapsacks if every knapsack maintains a data load within its balancing capacity. Every reducer is considered as a knapsack, the balancing capacity of reducer i is, sh_i, as shown in Eq. (6). Let's assume there are C data clusters, the frequency of keys in $cluster_j$ is key_j, where $(j = 1, 2, .., C)$. These data clusters have to be partitioned into R reducers, the data share of reducer i that satisfy the balancing capacity is sh_i, then we can get the balancing partitioning based on Eq. 7.

$$\begin{aligned} \min_{T_{ij}} \ & \sum_{i=1}^{R} \left| \left(\sum_{j=1}^{C} \left(key_j \times T_{ij} \right) \right) - sh_i \right| \\ \text{s.t.} \ & \sum_{i=0}^{R} \mathbf{T}_{ij} = 1, \text{for all } j = (1, .., C) \\ & \mathbf{T}_{ij} = 0 \ or \ 1, \text{for all } i = (1, .., R) \\ & and \ j = (1, .., C) \end{aligned} \tag{7}$$

Since the data clusters vary in their sizes, and one data cluster cannot be split into two reducers, then the partitioner cannot guarantee that the reducer receives data load that equal its data share. However, the data load on a reducer is either exceed its share or lower than it. The partitioning algorithm minimize this difference as much as possible in order to give better data load balancing among the reducers. As shown in the above minimizing equation, the balancing partitioner minimizes the absolute difference between the load on the reducer and the balancing capacity of the reducer.

To guarantee the best-balanced distribution, the algorithm sorts the data clusters in descending order according to the number of keys in each data cluster. The algorithm selects the first data cluster in the sorted array (data cluster with largest size), assigns it to the reducer with the largest balancing capacity , and update the remaining capacity of the reducer.

In the second round, the algorithm selects the second data cluster in the array (second largest size data cluster), assign it to the biggest capacity reducer, update the remaining capacity of the reducer, and so on until it reaches the last data cluster (smallest data cluster). Note that, with the progress of the distribution algorithm, the sizes of data clusters become smaller, and the capacity of reducers get lower, and so on.

The reason of leaving the smaller data clusters to the end of distribution is because the small size data clusters are easier to be distributed without effecting the overall balancing of data load among reducers.

The data clusters assignment algorithm for heterogeneous reducers is shown in Algorithm 3. Lines $2 - 4$ initiate the reducers competition array. The initial capacity of every reducer is the data share of reducer calculated in Eq. (6). The value that represents capacity of reducer in the competition array is decreased every time the reducer has been assigned a data cluster. Line 5 sorts the sizes of data clusters in descending order as mentioned earlier in the Algorithm 2.

In line 9, for every data cluster, the algorithm checks if the mapper node that produce the keys of this data cluster is Mapper/Reducer node. When such a case exists, and the Mapper/Reducer node produces more than half of the size of this data cluster, during the map phase, the data cluster is assigned to the Mapper/Reducer node for reduce

task, as discussed earlier in homogeneous environment. Line 12 is the normal mode of distribution. The function elected_R( ) returns the index of reducer with the minimum data load. Line 14 is for assigning the data cluster to the selected reducer. Line 15 updates the capacity of the selected reducer, and so no until the algorithm reaches the last data cluster in the array (smallest data cluster).

Algorithm 3: Data clusters assignment algorithm-heterogeneous

Input: **Input**: A collection of **R** reducers.
$\mathbf{sh_i}$ The share of the intermediate data of $reducer_i$ that satisfying balancing capacity. Calculated using Eq. (6)
C data clusters to be distributed, which are numbered $Cluster_0$ through $Cluster_{C-1}$.
Key[i] is the number of keys in data $Cluster_i$
Result: $T[1, 2, ..R][1, 2, .., C]$

```
1  begin
2      for i = 1 to R do
3          reducer_compete [i] = sh[i];
4      end
5      Sorted_key[ ] = sort_descending(Key[ ]);
6      Index_Sorted_key[ ]=Index of original order
       ofsort_key[ ];
7      for i = 1 to C do
8          j = Index_Sorted_key[ i];
9          if (keys of Cluster_j are the Map output of
           Mapper/Reducer node) && (this node
           produces more than half of data cluster j)
           then
10             z = Index(Mapper/Reducer node);
11         else
12             z = elected_R
               (reducer_compete[1,2,..R])
13         end
14         T[z] [j] = 1;
15         reducer_compete[ z] =
           reducer_compete[ z] −Key[ j];
16     end
17     Return T[1, 2, ..R][1, 2, .., C]
18 end
```

In both homogeneous and heterogeneous reducers environments, during the sampling phase, the selected sample size controls the accuracy of representation of input data in the sample. If the sample size is small, many keys in the input data may not appear in the sample. The smaller the sample size, the higher probability the key is not appearing in the sampling phase. Therefore, it is very normal situation when many keys are not presented in the sample.

BDCP is designed for partitioning the keys that appears in the sampling phase. Moreover, it calculates the actual size of data based on the resulted data sample. To solve this issue for those keys that do not appear in the sample, *BDCP* applies the default HashPartitioner on those data clusters because their size is very small in the actual input data and do not cause reducers data skew.

The partitioning process using *BDCP*, adds extra phase (sampling phase) to *MapReduce* job. The execution time of sampling phase cannot be overlapped with the execution times of the other phases. Sampling phase makes the actual map tasks on input data starts later than the actual job start time. This delay should guarantee minimizing the reduce phase time, and slightly decreasing the shuffle phase time. As illustrated in the Fig. 6.

Evaluation

To test the performance of the proposed strategy, homogeneous and heterogeneous Hadoop cluster environment have been prepared. Practical MapReduce application jobs were chosen to evaluate the performance of *BDCP*. The crucial factor is to minimize the job execution time. The performance of the suggested algorithm is examined by comparing the execution time of reduce phase using this algorithm against other algorithms. Two different types of benchmarks with synthetic and real-world datasets with different data skew rate are used to evaluate *BDCP* in homogeneous and heterogeneous Hadoop clusters. We compared *BDCP* with the default Hadoop HashPartitioner, and Range partitioner [9] in same experiment environment.

Hadoop HashPartitioner is the default mechanism in Hadoop environment which can obtain a good performance only when the (*Key*, *Value*) pairs are distributed uniformly. Range partitioner is a widely used algorithm of partition distribution in which the intermediate (*Key*, *Value*) pairs are sorted by key first, and then the pairs are assigned to reduce tasks according this key range sequentially. Range is one of the algorithms that can improve the data balance among reduce tasks.

The performance of default Hadoop hash-partitioner, range partitioner, and BDCP have been compared based on the reduce execution time in order to verify the effect of intermediate data placement. Heavy MapReduce jobs that process large amounts of input data, and generate large intermediate data are implemented. In order to ensure accuracy, each group of experiments has been executed at least 10 times, and the mean value has been used as result. We implemented our system on Hadoop YARN version 2.9.2. During the sampling phase, we choose 10% of input data as the sample size. We implemented the experiments on homogeneous cluster environment, then we implemented the same experiments on heterogeneous cluster environment.

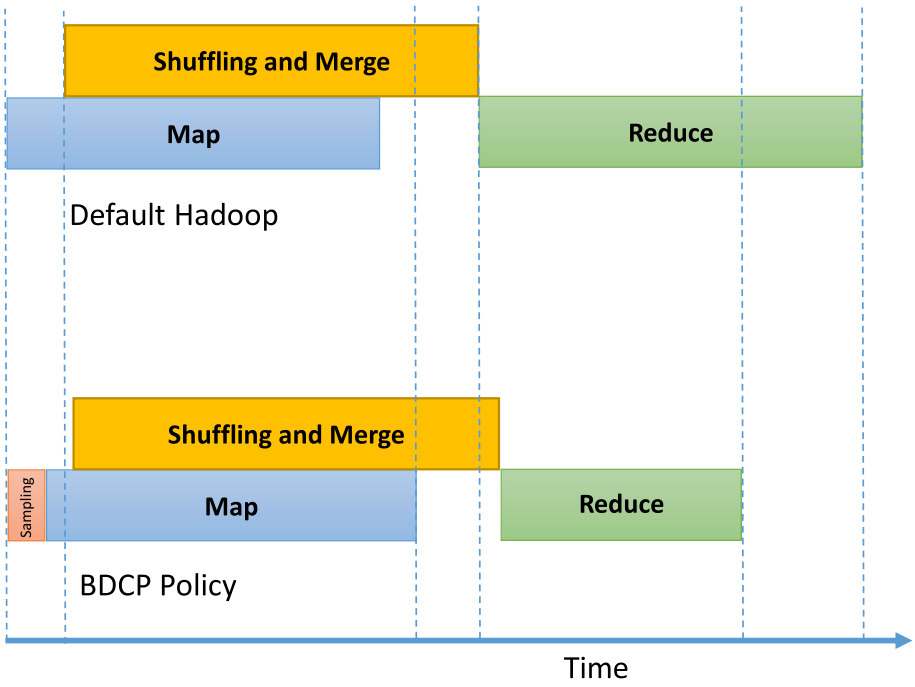

Fig. 6 MapReduce phases for Default Hadoop and BDCP

Homogeneous cluster experiments

The experiments are conducted on a Hadoop YARN cluster consists of 20 physical machines connected on single switch with 1 Gbps network bandwidth, installed with Ubuntu 14.10. with 8 cores 2.53 GHz processors, 16G memory, 1TB hard disk. The proposed system has been implemented and evaluated by running different types of benchmarks.

Word count benchmark testing

Three algorithms are compared under Word Count benchmark. Word Count job counts the number of each key in a file and produces an output file containing all keys and their frequencies. a heavy MapReduce job has been used to processes large amount of input data. The input data is split into blocks in HDFS. Each block is processed line by line to count the number of keys. Word Count job is suitable job to test the proposed algorithm because it generates large intermediate data .

Figure 7 shows the reduce phase execution time of word count job on 6 GB of data file, the skew degree of the data used ranges from 0.1 to 1.1. As shown in the figuer, as the data skew increases the processing time gets larger because of the data skew leads to reducers skew especially with using the HashPartitioner. At the beginning, the execution times is relatively low. The increasing of data skew has big impact on the reduce phase execution time using Hadoop HashPartitioner and lower impact on Range algorithm, while BDCP algorithm mitigates this impact and shows slightly increasing in processing time as the data skew increases.

Figure 8 shows the reduce execution time of word count job on files with data sizes, 2 GB, 4 GB, and 6 GB. The skew degree of the data used is 0.1. Even though the data skew is low, the increasing of file size makes the reduce phase execution time increases with different ratios depending on the used partitioner. Figure 9 shows the reduce execution time of word count job on file with data sizes, 2 GB, 4 GB, and 6 GB and the skew degree of the data used is 1.1. Because the data skew is high, the reduce phase execution time is high even when the file size is not large. The increasing of file size makes the execution time longer, because the bigger data file the more intermediate data skew for the same input data skew degree.

Sort benchmark test

Sorting is a part of widely adopted benchmarks for parallel computing [26]. Sort job, a reduce-input- heavy job, is used to test the proposed algorithm by processing input data with different data skew degrees. sort benchmark job has been implemented on file with size 6 GB of data, and the skew degree of the data used ranges from 0.1 to 1.1.

The reduce phase execution time of BDCP is shorter than Hadoop HashPartitioner and Range when processing

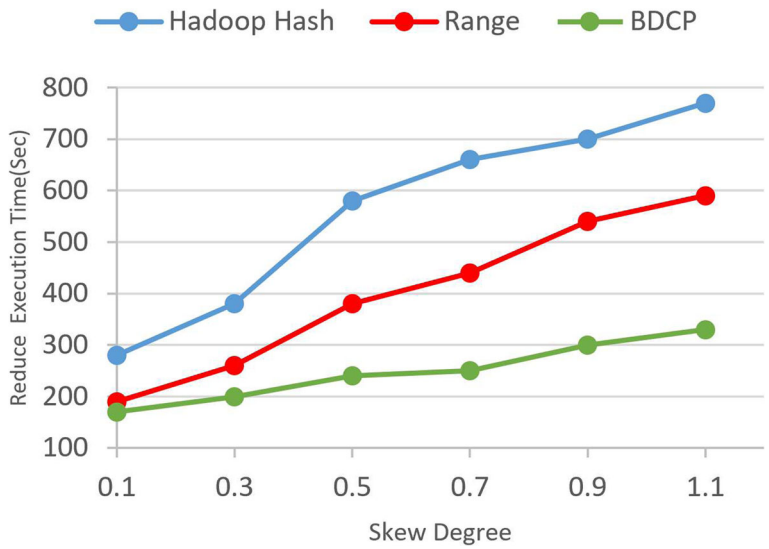

Fig. 7 Word count job on a file of size 6 GB, and 0.1 to 1.1 skew degree

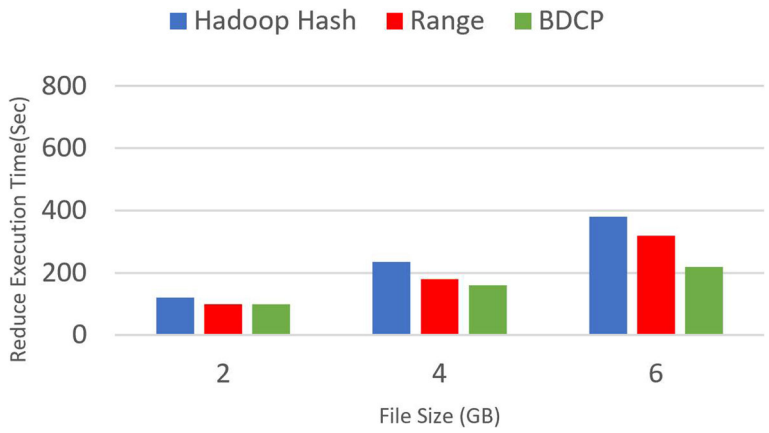

Fig. 8 Word count job on files of sizes, 2, 4, and 6 GB. skew degree is 0.1

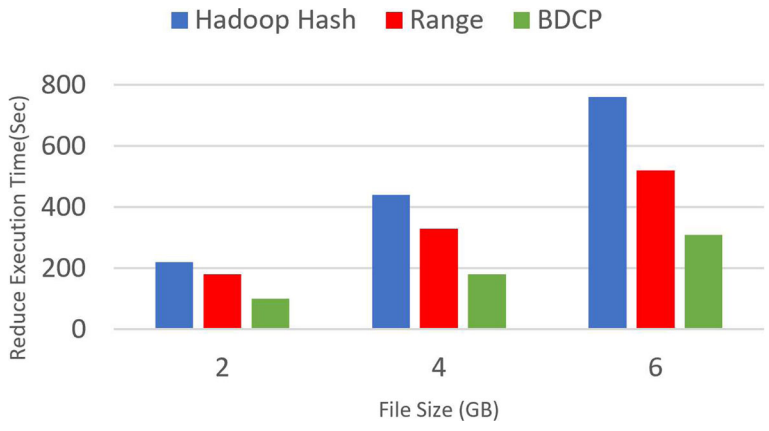

Fig. 9 Word count job on files of sizes, 2, 4, and 6 GB. skew degree is 1.1

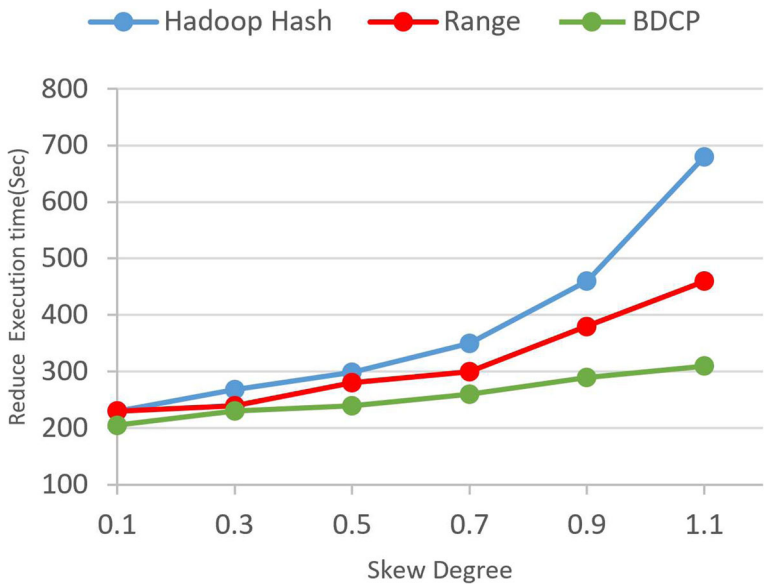

Fig. 10 Sort job on a file of size 6 GB, and 0.1 to 1.1 skew degree

the data with high skew rate. if the data skew rate is lower than a certain value, BDCP performance is better but it is close to the performance of the other two algorithms. While BDCP performs much better with the increase of data skew. As shown in Fig. 10, when data skew degree is less than 0.40, the Hadoop HashPartitioner has an execution time closer to the other two algorithms because of its even partitions of intermediate data.

When the skew becomes more than 0.3 BDCP starts to outperform the Hadoop HashParitioner and Range algorithms.

When both the data skew degree and file size are small, the reduce phase execution times for the three algorithms are relatively low. However, the increasing of file size with the same data skew degree makes the execution time slightly higher. However, *BDCP* has lower execution time than other two algorithms with the increase of the file size. Figure 11 shows the reduce phase execution time of Sort jobs on files with data sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 0.1.

Using higher data skew for different sizes of data files shows that BDCP highly outperforms both algorithms. With the increasing of data file for the same (high) data skew, the execution time of reduce phase of BDCP becomes less than half of the reduce phase execution time using HashPartitioner, and less than 0.6 of the reduce phase execution time using Range partitioner. Figure 12 shows the reduce execution time of Sort jobs on files with

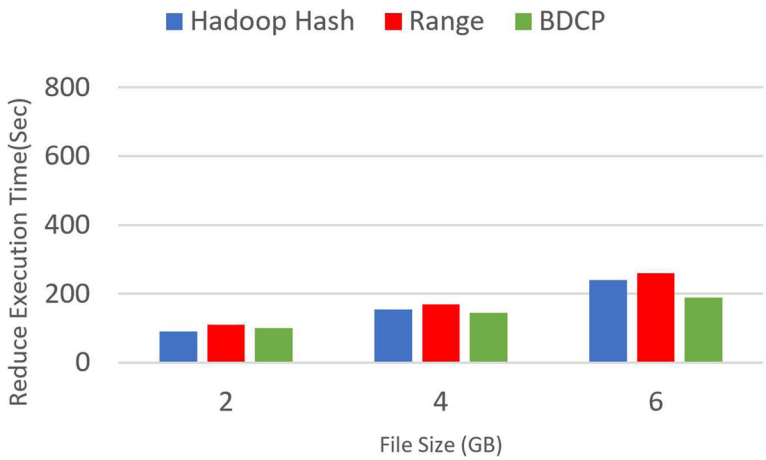

Fig. 11 Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 0.1

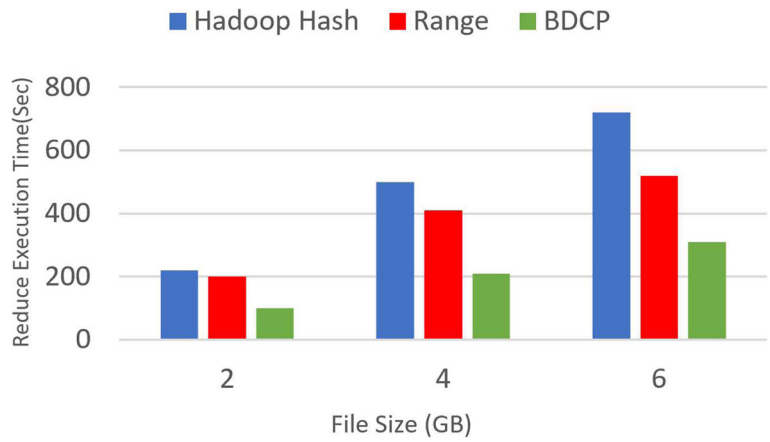

Fig. 12 Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 1.1

sizes, 2 GB, 4 GB, and 6 GB of data, the skew degree of the data used is 1.1.

Heterogeneous cluster experiments

In order to implement the partitioning algorithm in a heterogeneous cluster environment where the reducers vary in their available resources, experiments are conducted on a Hadoop YARN cluster consists of 20 physical machines connected on single switch with network bandwidth of 1 Gbps installed with Ubuntu 14.10, as following:

- 10 machines, 8 core 2.53 GHz processors, 16G memory.
- 5 machines with 4 core 3.4 GHz processor, 8G memory.
- 5 machines with 4 core 2.7 GHz processors, 4G memory.

The proposed system has been implemented and evaluated by running different types of benchmarks. The data skew and the variation in DataNodes processing capabilities are the main two reasons that cause straggler tasks.

In this experimental environment these two factors are exist. In this experiment the impact of these factors on the reduce phase running time is examined by running different types of benchmarks.

Word count benchmark

Heterogeneous Hadoop cluster with the mentioned configuration produces unbalanced distribution of intermediate data load among reducers nodes which results in increasing the processing time of reduce phase especially in HashPartitioner. However the increasing of data skew Aggravates the problem of balancing the data load among reducer nodes. Figure 13 shows the reduce execution time of word count jobs on a files with 6 GB of data, the skew degree of the data used ranges from 0.1 to 1.1. The figure shows that the reduce phase processing time in BDCP is lower than the reduce phase processing time of the other two algorithms for all skew degrees.

Even though the processing times for the three algorithms are relatively low when the data skew is low, they are much higher than those of same job in homogeneous cluster environment. The reduce phase processing time increases with the increasing of data skew with the existing of variation of reducers processing capabilities. BDCP shows good mitigation for the increasing of data skew in heterogeneous cluster environment.

Figure 14 shows the reduce phase execution time of word count job with data sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 0.1. As shown the Figs. 13 and 14, even though when the data skew is low, BDCP is achieving better than Hadoop HashPartitioner and Range because it considers the variation of computing capabilities of the reducers. While HashPartitioner, and range took longer time than the time they took previously in homogeneous cluster environment. Figure 15 shows the reduce execution time of word count job on a file with data sizes, 2 GB, 4 GB, and 6 GB, on a heterogeneous Hadoop cluster, the skew degree of the data used is 1.1.

Both high data skew and variation in computing capabilities of the reducers result in higher reduce execution time for all algorithms. The results show that BDCP algorithm always has the shortest reduce time for different file sizes with high skew degrees. However the increasing of file size, for the same input data skew degree, increases the execution time of reduce phase.

Sort benchmark test

Sort benchmark job has been implemented on a file with 6 GB of data, the skew degree of the data used ranges from 0.1 to 1.1. Figure 16 shows the reduce execution time of

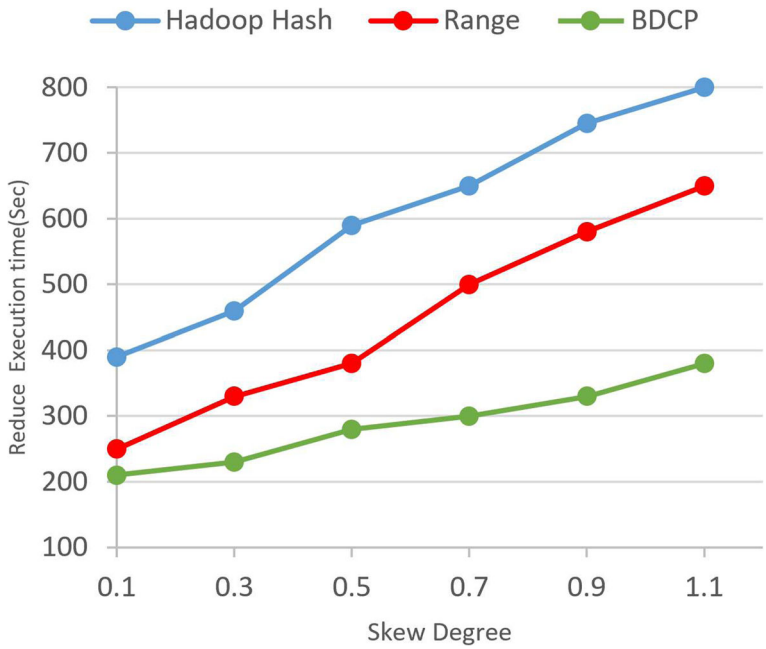

Fig. 13 Word count job on a file of size 6 GB, and 0.1 to 1.1 skew degree

the sort job. BDCP works in similar efficiency to Hadoop HashPartitioner and Range when the data skew rate is lower than 0.2, but it gives shorter execution time, while it performs better with the increasing of data skew. BDCP is much faster than Hadoop HashPartitioner and Range in processing the data with high skew rate.

Hadoop HashPartitioner of intermediate data does not consider the heterogeneity on the reducers, so it takes more time than the previous sort experiment on homogeneous environment. Figure 17 illustrates the reduce phase execution time of Sort job of file with sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 0.1. BDCP performs better than HashPartitioner and Range. The differences in reduce phase execution times for the three algorithms increase with the increasing of the file size for the same skew degree. However, the variations in reduce phase execution time are not large because the skew degree is low.

Figure 18 shows the reduce phase execution time of Sort jobs on files with data sizes, 2 GB, 4 GB, and 6 GB, the skew degree of the data used is 1.1. For this high skew degree, the reduce phase execution times for the three algorithms are relatively high and increase with the increasing of the file size. However, the variations in

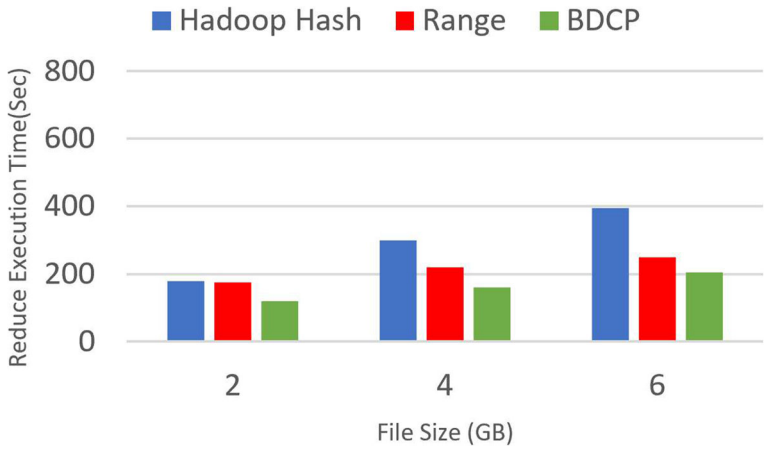

Fig. 14 Word count job on files of sizes, 2, 4, and 6 GB. Skew degree is 0.1

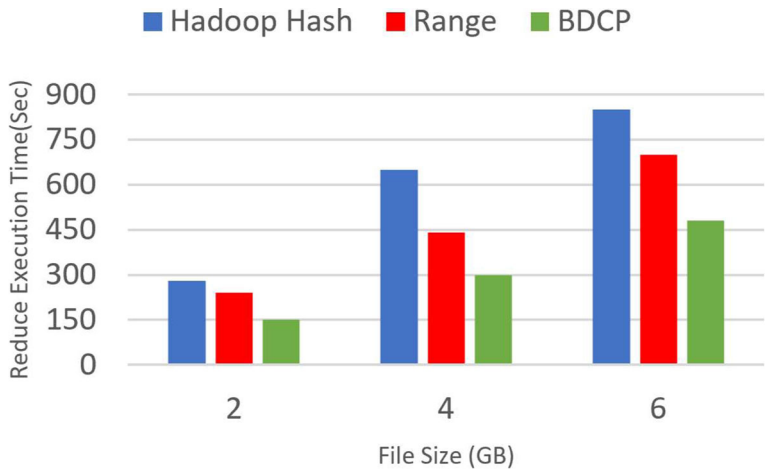

Fig. 15 Word count job on files of sizes, 2, 4, and 6 GB. Skew degree is 1.1

reduce phase execution time are large because the skew degree is high. BDCP outperforms HashPartitioner and range especially with the increasing of file size with high data skew.

To illustrate the timing of all the MapReduce job phases, a Word Count job on a file with size of 2 GB of data and the skew degree is 0.3 has been implemented using the same configuration of heterogeneous cluster environment. BDCP algorithm divides MapReduce job execution process into five phases, as shown in Fig. 19. The phases are represented on the figure are sampling, map, overlapped map and shuffle, shuffle, and reduce phase. Sampling phase is only used by BDCP algorithm. It can not be overlapped with map phase. In BDCP, map phase starts right after sampling phase.

Because the partitioning policy has been already generated after the sampling phase, shuffle phase starts whenever there is an output from the map phase. While the Hadoop HashPartitioner begins shuffling the map task outputs when 5% of map tasks have completed. The execution time of reduce phase is minimized in BDCP as shown in Fig. 19.

Conclusions

In this paper, we minimize the effect of the commonly known problem of skew data and mitigate straggler reduce

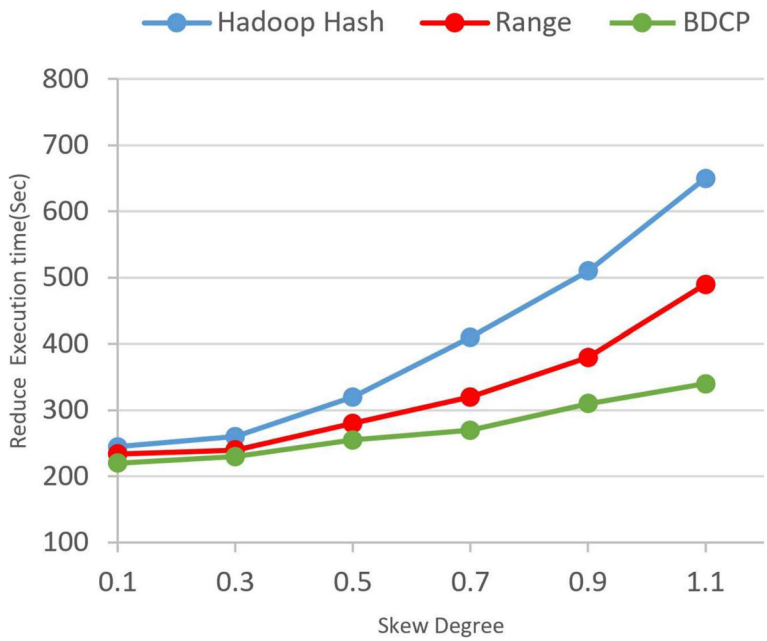

Fig. 16 Sort job on a file of size 6 GB, and 0.1 to 1.1 skew degree

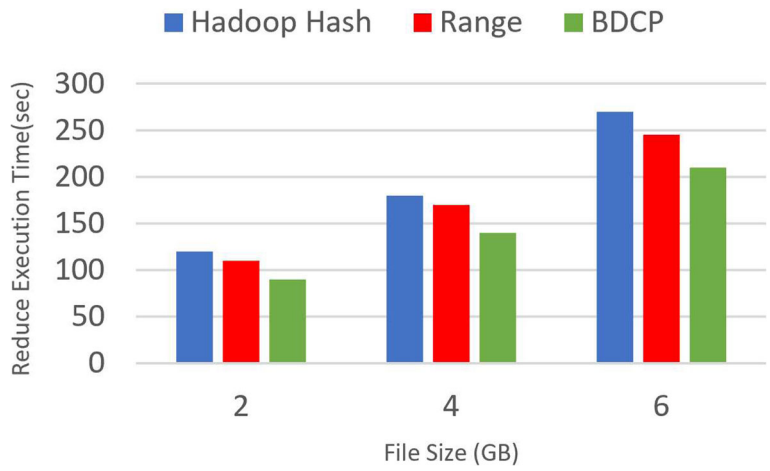

Fig. 17 Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 0.1

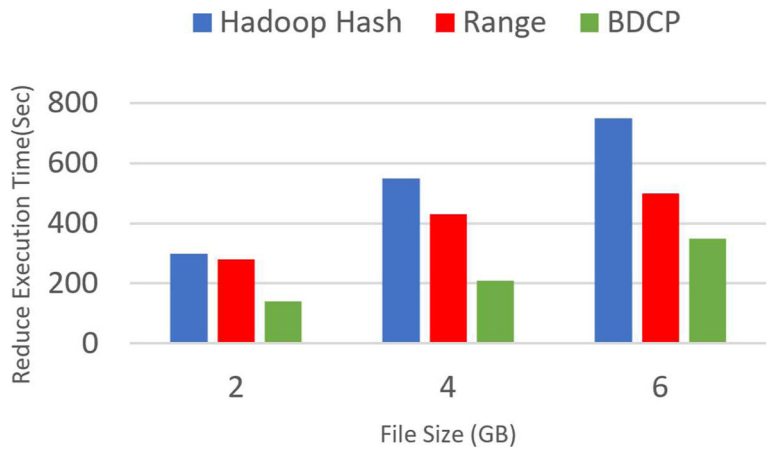

Fig. 18 Sort job on files of sizes, 2, 4, and 6 GB. Skew degree is 1.1

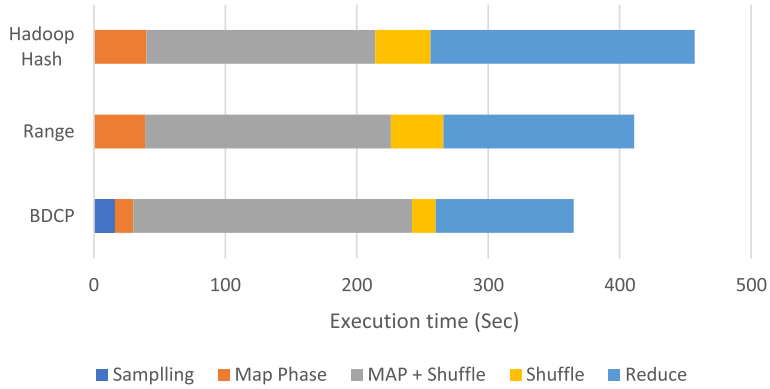

Fig. 19 MapReduce phases of Word Count job on a file of size 2 GB, and skew degree is 0.3

task by balancing the distribution of intermediate data to reducer. We propose *BDCP* algorithm, it adds a new phase to MapReduce job, it is the sampling phase. During the sampling phase, an estimation of the data distribution is calculated and feedback about reducer processing ability is received during the process of creating partitioning policy. We conduct extensive experiment on different data size and skew degree to evaluate the performance of *BDCP* compared with the Hadoop-Hash and Range algorithms. The simulation results indicate that *BDCP* give better reduce task completion time. In the future, we are planning to calculate network bandwidth between cluster nodes to add it as a factor to this algorithm.

Abbreviations

BDCP: Balanced Data Clusters Partitioner; HDFS: Hadoop Distributed File System; MAT: Main Assignment Table; QoS: Quality of Service; SRS: Simple random sample

Acknowledgements

We wish to thank the anonymous reviewers for their valuable feedback for improving the quality of the manuscript.

Authors' contributions

The research presented in this paper is part of the Ph.D. dissertation of the first author under the supervision of the second author. Both authors read and approved the final manuscript.

Authors' information

Ibrahim Adel Ibrahim received the Bachelor of Engineering degree in Computer Engineering in 2002 from Almustanseriah University/Collage of Engineering, and the MS degree in Computer Engineering in 2006 from the University of Technology. He is currently a Ph.D. student in Computer Engineering at the University of Central Florida, Orlando, Florida. His research interests include cloud computing, data-intensive computing, and computer networks. Mostafa Bassiouni received his BS and MS degrees in Computer Science from Alexandria University and received the Ph.D. degree in Computer Science from the Pennsylvania State University in 1982. He is currently a professor of Computer Science at the University of Central Florida, Orlando. His research interests include computer networks, distributed systems, real-time protocols and concurrency control. He has authored and coauthored over 210 papers published in various computer journals, book chapters and conference proceedings. His research has been supported by grants from ARO, ARPA, NSF, STRICOM, PM-TRADE, CBIS, Harris, and the State of Florida. He is an Associate Editor of the Computer Journal- Oxford University Press, Editor-in-Chief of Electronics-MDPI, and an Editorial Board Member of four other journals. He has served as member of the program committee of several conferences, as the program committee chair of CSMA'98 and CSMA'2000 and as the guest co-editor of a special issue of the Journal of Simulation Practice and Theory, 2002.

Funding

Not applicable.

Availability of data and materials

Not applicable.

Competing interests

The authors declare that they have no competing interests.

Author details

[1]Department of Computer Engineering, University of Central Florida, Florida, USA. [2]Department of Computer Engineering, University of Technology, Baghdad, Iraq. [3]Department of Computer Science, University of Central Florida, Florida, USA.

Received: 21 May 2019 Accepted: 26 September 2019
Published online: 07 February 2020

References

1. MapReduce: Official Apache Hadoop Website. http://hadoop.apache.org. Accessed 14 Feb 2019
2. Wu H (2016) Big data management the mass weather logs. In: International Conference on Smart Computing and Communication. Springer. pp 122–132
3. White T (2009) Hadoop, "The Definitive Guide (1'st ed.)"
4. Subramanian V, Wang L, Lee E-J, Chen P (2010) Rapid processing of synthetic seismograms using windows azure cloud. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science. IEEE. https://doi.org/10.1109/cloudcom.2010.110
5. Chen Q, Yao J, Xiao Z (2015) Libra: Lightweight data skew mitigation in mapreduce. IEEE Trans Parallel Distrib Syst 26(9):2520–2533
6. Zhang F, Cao J, Khan SU, Li K, Hwang K (2015) A task-level adaptive mapreduce framework for real-time streaming data in healthcare applications. Futur Gener Comput Syst 43:149–160
7. MapReduce Job. Word Count. http://spark.apache.org/examples.html. Accessed 27 Apr 2019
8. Lee D, Kim J-S, Maeng S (2014) Large-scale incremental processing with mapreduce. Futur Gener Comput Syst 36:66–79
9. Range Partitioner, [EB/OL]. http://spark.apache.org/docs/1.3.0/api/java/org/apache/spark/RangePartitioner.html. Accessed 11 Apr 2019
10. Kwon Y, Balazinska M, Howe B, Rolia J (2012) Skewtune: mitigating skew in mapreduce applications. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data. ACM, Scottsdale. pp 25–36
11. Hassan MAH, Bamha M, Loulergue F (2014) Handling data-skew effects in join operations using mapreduce. Procedia Comput Sci 29:145–158
12. Karapiperis D, Verykios VS (2015) Load-balancing the distance computations in record linkage. ACM SIGKDD Explor Newsl 17(1):1–7
13. Vu L, Alaghband G (2015) A load balancing parallel method for frequent pattern mining on multi-core cluster. In: Proceedings of the Symposium on High Performance Computing. Society for Computer Simulation International, Alexandria. pp 49–58
14. Li J, Liu Y, Pan J, Zhang P, Chen W, Wang L (2017) Map-balance-reduce: an improved parallel programming model for load balancing of mapreduce. Futur Gener Comput Syst. https://doi.org/10.1016/j.future.2017.03.013
15. Xu Y, Zou P, Qu W, Li Z, Li K, Cui X (2012) Sampling-based partitioning in mapreduce for skewed data. In: 2012 Seventh ChinaGrid Annual Conference. IEEE. https://doi.org/10.1109/chinagrid.2012.18
16. Tang Z, Zhang X, Li K, Li K (2018) An intermediate data placement algorithm for load balancing in spark computing environment. Futur Gener Comput Syst 78:287–301
17. Ibrahim IA, Bassiouni M (2017) Improving mapreduce performance with progress and feedback based speculative execution. In: 2017 IEEE International Conference on Smart Cloud (SmartCloud). IEEE. https://doi.org/10.1109/smartcloud.2017.25
18. Ananthanarayanan G, Ghodsi A, Shenker S, Stoica I (2013) Effective straggler mitigation: Attack of the clones. In: Presented as Part of the 10th {*USENIX*} Symposium on Networked Systems Design and Implementation ({*NSDI*} 13). USENIX, Lombard. pp 185–198
19. Zaharia M, Konwinski A, Joseph AD, Katz RH, Stoica I (2008) Improving mapreduce performance in heterogeneous environments. Osdi 8:7
20. Xie J, Yin S, Ruan X, Ding Z, Tian Y, Majors J, Manzanares A, Qin X (2010) Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW). IEEE, Atlanta. pp 1–9
21. Lin C, Guo W, Lin C (2013) Self-learning mapreduce scheduler in multi-job environment. In: 2013 International Conference on Cloud Computing and Big Data. IEEE. pp 610–612. https://doi.org/10.1109/cloudcom-asia.2013.95
22. Ibrahim IA, Dai W, Bassiouni M (2016) Intelligent data placement mechanism for replicas distribution in cloud storage systems. In: 2016 IEEE International Conference on Smart Cloud (SmartCloud). IEEE. https://doi.org/10.1109/smartcloud.2016.23
23. Dai W, Bassiouni M (2013) An improved task assignment scheme for hadoop running in the clouds. J Cloud Comput Adv Syst Appl 2(1):23
24. Dai W, Ibrahim I, Bassiouni M (2016) A new replica placement policy for hadoop distributed file system. In: 2016 IEEE 2nd International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing (HPSC), and IEEE International Conference on Intelligent Data and Security (IDS). IEEE. pp 262–267. https://doi.org/10.1109/bigdatasecurity-hpsc-ids.2016.30

25. Dai W, Ibrahim I, Bassiouni M (2016) Improving load balance for data-intensive computing on cloud platforms. In: 2016 IEEE International Conference on Smart Cloud (SmartCloud). IEEE. https://doi.org/10.1109/smartcloud.2016.44
26. Khatami Z, Hong S, Lee J, Depner S, Chafi H, Ramanujam J, Kaiser H (2017) A load-balanced parallel and distributed sorting algorithm implemented with PGX.D. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). IEEE. https://doi.org/10.1109/ipdpsw.2017.30

Publisher's Note