A Novel Key Partitioning Schema for Efficient Execution of MapReduce Applications

Saeed Nasehi Basharzad
Computer Department
Sharif University of Technology
Tehran, Iran
saeednasehi@ce.sharif.edu

Seyed Morteza Nabavinejad
Computer Department
Sharif University of Technology
Tehran, Iran
Nabavinejad@ce.sharif.edu

Maziar Goudarzi
Computer Department
Sharif University of Technology
Tehran, Iran
Goudarzi@sharif.edu

***Abstract*— MapReduce and its open source implementation, Hadoop, are the prevailing platforms for big data processing. MapReduce is a simple programming model for performing large computational problems in large-scale distributed systems. This model consists of two major phases: Map and Reduce. Between these two main phases, partitioner part is embedded which distributes produced keys by Map tasks among Reduce tasks. When the amount of keys and their associated values, which are called intermediate data, is huge, this part has significant impact on execution time of Reduce tasks, and consequently, completion time of jobs. In this paper, we present a network and resource aware key partitioner to decrease the execution time of MapReduce jobs. Using sampling, our algorithm finds the distribution of keys in intermediate data. Then, considering aforementioned distribution, the amount of each key on each machine, the placement of Reduce tasks on machines and the network bandwidth between machines, our algorithm assigns keys to Reduce tasks to decrease the total execution time of job. Our experiments show that our approach can improve completion time of Reduce phase and job execution time by up to 52% and 31% respectively compared with Hadoop default partitioner and can find the solution within 8% of ideal partitioner.**

Keywords*— *MapReduce; Hadoop; Big data; Partitioner; Performance

I. INTRODUCTION

Recently, the volume of produced data in the world has been increasing significantly. Due to the increasing spread of Internet in everyday life, data production has become so simple because of the use of social networks, search engines and other similar reasons. For instance, in just one hour, users upload on average over 72 hours video on YouTube. According to the report of International Corporation on data, the volume of generated and copied data in 2011 was 1.8 ZB, and it will be doubled every two years [1].

This rapid growth in data production, led to invention of Big Data concept. To process Big Data, various frameworks and programing paradigms are proposed. MapReduce, which is a simple programming paradigm [2], is one of the prevailing options for processing Big Data in large-scale clusters.

Apache Hadoop [3] is an open source implementation of this model and is a framework that makes processing of large amount of data on a cluster of servers possible.

Breaking input data into small chunks and distributing these chunks in the cluster is the manner of Hadoop for processing large amount of data. Each Map task processes a chunk of data and produces intermediate data. Then, Reduce tasks process intermediate data to generate final result. Reduce is composed of three subsections; Shuffle, Sort and Reduce [4]. Shuffle is responsible for distributing generated key-value pairs by Map tasks among Reduce tasks. To do so, it uses a function called partitioner to assign key-value pairs to Reduce tasks [5].

Since MapReduce executes several tasks simultaneously, if the execution time of tasks is uneven, some tasks act as stragglers [6] and elongate execution of jobs, which causes resource waste. To prevent this issue, it is desirable that execution time of tasks be as close as possible. Various reasons may cause late or early completion of a task, such as processing power of cluster's nodes and their network bandwidth. The distribution of keys between tasks can also have a significant impact on completion time of Reduce tasks.

If partitioner function assigns a huge amount of key-value pairs to one Reduce task without considering the computing resources and network bandwidth of machine that the task is placed on, task would take longer than others to complete, and hence, the total execution time of job will increase. Uneven distribution of key-value pairs among Reduce tasks that can lead to tasks with different execution time, is a well-known issue called data skew.

Early Shuffling feature in Hadoop makes it possible for Shuffle phase to start before all the Map tasks are finished. In other words, it helps breaking the barrier between Map and Reduce phase and overlapping execution of Map and Reduce tasks, and consequently, improving the total completion time of jobs. Using *slowstart* parameter in Hadoop, one can determine the percentage of Map tasks that need to be finished before Shuffle phase can begin. Later in Related Work section, we will see that some of the current proposed approaches cannot support this feature and need all the Map tasks to be finished before Shuffle phase. Our proposed approach, however, can tackle this problem.

Hadoop default partitioner (see section 3) is a blind algorithm that only uses a hash function to distribute key-values to Reduce tasks. Hence, it is possible that several key-values with high frequency are assigned to one task while the other ones with negligible frequency are assigned to another.

A large body of research [6-10] has tried to address the data skew problem. Some of them [8-9] have proposed new partitioning functions to mitigate the uneven distribution of key-value pairs among Reduce tasks. Other approaches are also proposed that consider the computing power of machines, in addition to distribution of key-value pairs, when assigning keys to Reduce tasks [6][10]. However, none of them has considered distribution of key-values, computing power of machines, and network bandwidth between machines simultaneously.

In this paper we propose a new partitioner algorithm that considers locality, communication bandwidth and processing power of machines when distributing key-values among Reduce tasks. At the beginning, it uses a simple sampling method to estimate the distribution of keys in input data. Then, considering the placement of Map tasks and consequently the volume of key-value pairs generated on each machine, we calculate the execution and transfer time of each key in all the Reduce tasks considering the computing power and network bandwidth of machine that the Reduce task is placed on. The main objective of this algorithm is to make all the Reduce tasks to complete almost at the same time and avoid some of them act as stragglers and elongate the total completion time of job. Our major contributions in this work are as follows:

- Considering network bandwidth and processing power of resources simultaneously to decrease execution time of Reduce tasks.
- Including the locality of generated key-values on different machines when assigning keys to Reduce tasks
- Being compatible with early shuffling feature in Hadoop

The rest of the paper is organized as follows: Section 2 discusses the related works. In Section 3 basic background information for following this paper and motivation beyond this research are provided. An illustrative example to show the proposed algorithm's effectiveness and proposed approach are presented in Section 4 and 5. In section 6 we talk about simulation results. At last in section 7 of this paper we explain the conclusion of this paper.

II. Related Works

A large body of research has studied data skew [6-10]. LEEN approach introduced in [7] considers locality of key-values to improve the performance of default Hadoop partitioner through reducing the amount of key-values transferred in shuffle phase. It makes decision when all the Map tasks are finished, and hence, early shuffling is not supported by this algorithm. Another disadvantage of this algorithm is that it considers only homogenous clusters. LARTS algorithm presented in [8] improves LEEN approach by introducing three different of localities: node locality which means to have keys and Reduce tasks on the same machine, rack locality which means to have keys and Reduce tasks on the different machines but on the same rack, and finally off locality to have keys and Reduce tasks on different machines in different racks. However, it has the same issues as LEEN such as not attention to processing power of machines, and not supporting heterogeneous clusters. If the Reduce task is a slow task, attention to locality decrease the transferring time but execution time increases. It means that there is always a tradeoff between heterogeneity and fairness [9]. Libra approach [6] addresses the shortages of LEEN and LARTS and supports early shuffle and heterogeneous clusters. However, it does not consider network bandwidth. Ignoring network bandwidth can degrade performance, especially in networks with high bandwidth fluctuation. In [10] SkewTune algorithm is presented. This algorithm identifies one task that takes longer than usual and transfers the remaining data to another node. When a node in the cluster becomes idle, SkewTune identifies the task with the greatest expected remaining processing time. The unprocessed input data of this straggling task is then proactively repartitioned in a way that fully utilizes the nodes in the cluster and preserves the ordering of the input data so that the original output can be reconstructed by concatenation.

Our proposed method considers locality, network bandwidth and processing power to make more efficient decision. Moreover, our proposed approach can support early shuffling because it does an offline profiling before beginning of job, and hence, further improve total job completion time.

III. Background And Motivation

A. *MapReduce*

As we mentioned before, MapReduce is a programming paradigm [11] for implementing problems related to Big Data processing in large scale distributed systems. This model has been proved to gain high performance in distributed computing and Big Data processing [12]. This paradigm is consists of two main phases: Map and Reduce which we describe them blow.

Map. When Hadoop copies the input data to its file system (Hadoop Distributed File System, HDFS), it splits the data into data chunks or data blocks with same size (default size in Hadoop 1 is 64MB and in Hadoop 2 is 128MB). After that, for processing data blocks, Hadoop creates Map tasks. Each Map task is responsible for processing one data block. Hence, the number of Map tasks is equal to the number of data blocks. The output of each Map tasks is key-value pairs that are called intermediate data. Generated intermediate data by each Map task is stored on the local machine that executed that Map task.

Reduce. The duty of Reduce phase is merging the intermediate data and producing final result. As we mentioned before, this phase has three sub phases; Shuffle, Sort and Reduce.

Shuffle. In shuffling phase intermediate data is transferred to the defined Reduce task. This act is done by using an algorithm that is called partitioner [13]. Partitioner is an algorithm that assigns each key to a certain Reduce task.

Partitioner. As we mentioned earlier, distributing keys and values is the duty of partitioner. In Hadoop, the default partitioner is called hash partitioner that uses a hash function as below:

$$\textit{Reduce task number} = h(Key) \,\%\, R$$

That h is a hash function and R is the number of Reduce tasks. In this algorithm the Reduce task that should process a certain key is determined by hash function. Although, this partitioner does not need much time to respond, it distributes the keys regardless of the size and number of keys passed to a task. Hence, it may result in some Reduce tasks with heavy load which increases the key transfer and execution time of tasks and ultimately increases the overall run time. Moreover, this hash function completely ignores the computing power and network bandwidth of machines, and hence, is unable to tackle issues arisen by heterogeneous clusters.

Sort. This phase sorts the intermediate data in each Reduce task [14].

Reduce. At the third sub phase, Reduce tasks read the intermediate data and user defined Reduce algorithm is performed on them to produce final result [14].

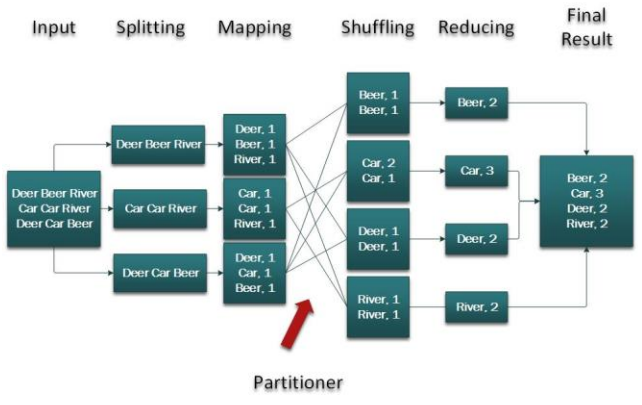

Fig. 1. WordCount program execution in MapReduce programming model

In figure 1, the execution flow of a WordCount program using MapReduce is shown. At first step, the input data is received and divided into equal parts. Then, in the Map phase, map function processes input data and generates intermediate data. Then shuffle step is executed. It should be noted that all key-value pairs with same key must be assigned to the same Reduce task. Finally, the output data of Reduce tasks are merged to give final result.

B. *Motivation*

As mentioned before, partitioner has a significant effect on processing time of Reduce phase. In the absence of proper distribution of key-values, data skew problem happens and performance of cluster will be dropped significantly in term of execution time and throughput and processing resources may be wasted. In figure 2, execution time of Reduce tasks for word count application in a cluster consists of 130 Reduce tasks is shown. Each vertical line represents the execution time of a Reduce task.

As you can see, in this example the time of execution for Reduce task with maximum run time is about 5 times greater than execution time of Reduce task with minimum run time. This figure indicates that partitioner must consider the key distribution and processing power in intermediate data. The lack of attention to this fact may cause latency in execution of application and wasting of resources.

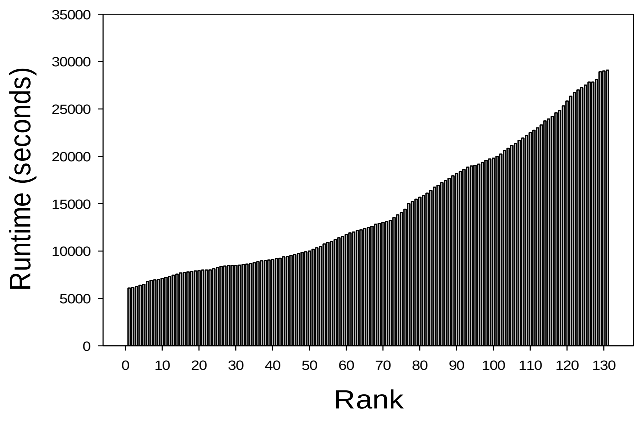

Fig. 2. Difference in the execution time of Reduce Tasks in Hadoop

IV. Illustrative Example

In this section, we provide an example to illustrate the necessity of a resource and network aware partitioner. Suppose that processing power and network bandwidth for four machines that execute the Reduce tasks is as table 1. Also assume that there is just one Reduce task on each machine.

TABLE I. Processing power and Transfer time of each machine for sample keys

Machine number	Processing power in time unit	Communication time in time unit
M1	1 Key-Value	1 Key-Value
M2	2 Key-Value	2 Key-Value
M3	2 Key-Value	3 Key-Value
M4	3 Key-Value	1 Key-Value

Now suppose that the input data includes keys K1 to K7 with the specifications provided in table 2. The two right columns in Table 2 shows the machines proposed for each key by two methods: our approach and default partitioner of Hadoop.

TABLE II. Sample keys and their partitioning to machines

Key	Number	Local Keys	Result of proposed partitioner	Result of default partitioner
K1	4	1 Key: M1, 1Key: M2, 2 Key: M3	M3	M1
K2	1	1 Key: M2	M2	M2
K3	2	2 Key: M1	M1	M3
K4	2	2 Key: M3	M3	M1
K5	3	2 Key: M2, 1 Key M3	M2	M4
K6	1	1 Key: M4	M4	M1
K7	3	1 Key: M1, 2 Key M4	M4	M1

The result of this example is provided in table3. As you can see, when we use default partitioner task No. 1 takes much more time than other tasks and act as a straggler. However, our approach prevents the straggler by proper distribution of keys among tasks. As you can see, our method assigns just K3 to Reduce task No. 1 while default partitioner assigns K1, K4, K6 and K7 to this task. This distribution decreases the execution time of this task from 16 units to 3 units.

TABLE III. Execution time of tasks using proposed and default partitioners

Task (Machine)	Task finishing time using Hadoop default partitioner	Task finishing time using proposed partitioner
1	16 time units	3 time units
2	1 time unit	2.5 time units
3	1.67 time units	2 time units
4	4 time units	2 time units

In this paper we propose a new network and resource aware partitioner. This algorithm considers the network bandwidth, locality and processing power in heterogeneous environments. We do not try to just decrease the volume of transferred data in shuffle phase, but to balance and decrease the time of computation and communication through efficient key partitioning.

V. Algorithm Definition

The workflow of proposed method is shown in Figure 3. As can be seen, we need a new part of processing called sampling. After getting sample of whole data, sample data is uploaded to HDFS. This action makes algorithm to have less overhead compared with preprocessing. In execution time of main algorithm, the same algorithm will be run on sample data. Then, some intermediate data is generated. Partition manager, by considering this data and the given algorithm, makes a decision for all keys.

The larger size of sample makes the better and more accurate estimation of distribution of keys. However, sampling time and running time will be increased for larger samples. Appropriate amount for sampling is 10% of volume of processing data according to [6]. Our sampling strategy in this paper is getting one megabyte of each 10 megabytes of data.

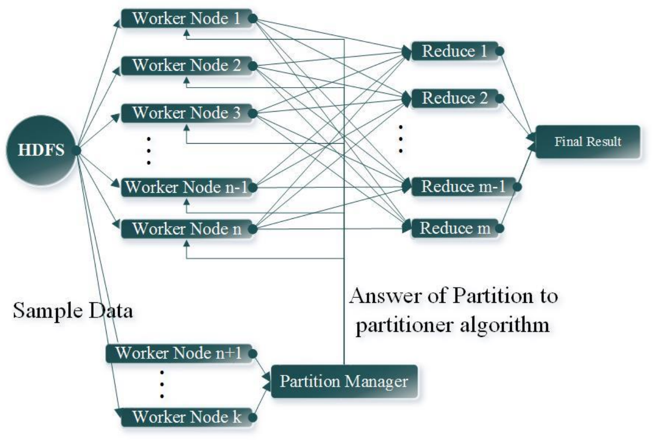

Fig. 3. Workflow of processing

After processing sample data and estimating distribution of keys in intermediate data for each machine, we make decision about key-value assignment to Reduce tasks. Our final goal is assigning the keys to Reduce tasks by considering the execution and transfer time, as well as locality of key-values, so that all the Reduce tasks will finish in about the same time. Inputs of our algorithm are as follows: set of existing keys on sample data on each machine, bandwidth of machines, processing rate for each machine (explained in next paragraph).

Processing rate is the relative number which determines the number of keys can be processed in time unit by a machine divided by the number of keys can be performed by the weakest machine in the cluster. For example, if the weakest machine takes one unit of time to perform Reduce algorithm on 100 keys, and other machine is able to execute them in 0.8 of time unit, the processing rate for the weakest machine is 1 and for the other machine is 1.25.

Then, we need to find processing unit. Processing unit is a number that shows the least number of keys in each range of keys that must be assigned to Reduce tasks. First we select the most repetitive key as processing unit. If this number is less than the 1/(Number of Reduce Task) of summation of all keys we select 1/(Number of Reduce Task) of summation of all keys as processing units. This action prevents from having huge number of ranges for partitioner phase. Multiplication of this number by processing rate of each machine (which is earned in previous step) shows the number of keys which must be performed by each machine and it makes sure if this number of keys is given to all machines, all machines will finish performing on those keys almost at the same time. Below algorithm is repeated until all the keys are assigned (you can see the pseudo-code in Algorithm 1):

- Select keys for each machine from sorted list of available keys. The summation of selected keys must be less than or equal to the multiplication of processing rate of each machine by processing unit (Line 6 of Algorithm 1).
- Then we calculate the amount of time required for executing and transferring all Reduce task according to the line 7-11 of Algorithm 1. Please note that transferring time is only for those keys which are not on this machine and should be transferred using network (locality). Then, in line 11 transferring time is calculated using network bandwidth.
- Now, according to the obtained information, we select a machine to assign selected keys to it. We select a machine that its current runtime plus the execution and transfer time of selected keys would be less than or equal of runtime of other machines. If there is no such machine, we select the one with minimum current runtime (Lines 12 – 17 of algorithm 1).
- Update the runtime of selected machine and delete selected keys from available keys list. Also assign this range of keys to selected machine.

Assuming that each Reduce task is assigned to one machine, the algorithm divides the keys into several ranges and assigns each range to a Reduce task. If we show the number of keys with *N*, and number of Reduce tasks with *R*, then the time complexity of our algorithm would be *O(N*R)*.

Algorithm 1 shows the pseudo code of proposed algorithm. Also in table 4 abbreviations which are used in algorithm 1 are explained.

TABLE IV. Abbreviations used in pseudo code

Variable	Explanation
ET_j	Execution time of Reduce task on j^{th} machine
EP_j	Processing rate in time unit on j^{th} machine
N	Number of keys in sample data
R	Number of Reduce tasks
$Keys_i$	Set of keys on i^{th} machine
All_Keys	Set of all keys
NB_j	Network bandwidth of j^{th} machine
TT_j	Transferring time to the j^{th} machine
V_{im}	Volume of i^{th} key that must be transferred to machine *m*
$EATj$	Execution and transferring time on j^{th} machine
$Total_i$	Busy time of i^{th} machine
Res	Set of keys and assigned Reduce Task number

Algorithm1. Network and resource aware partition algorithm

```
1: Input: EP, N , NB, Keys, All_Keys
2: Output: Res = {($k_1$,$k_2$,m) | Key K1 is beginning,$K_2$
       is end of range and m is assigned machine}
3: while available_keys is not empty:
4:    for m in machines:
5:       machine_capacity = processing_rate$_m$ * processing_unit
6:       Selected_key = select the first machine_capacity
              keys of available list
7:       for k in selected_key:
8:          $ET_m$ += $N_k$/$EP_m$
9:          $V_{km}$ = (1-key$_m$(k))/All_Keys(k)
10:         $TT_m$ += $v_{km}$ / $NB_m$
11:      end for
12:      for m in machines:
13:         $EAT_m$ = $ET_m$ + $TT_m$
14:         $Temp_i$ = $EAT_{ij}$ + $Total_i$
15:      end for
16:      m = Select the minimum Temp as selected   machine for this list
         of  keys, if there are some Temp with  minimum number, select
         machine with minimum cuurent runtime
17:      $Total_m$ = $EAT_m$ + $Total_m$
18:      del delected_key from available_key
19:      first_key = selected_key$_{first\_element}$
20:      last_key = selected_key$_{last\_element}$
21:     Add (first_key,last_key,m) to Res
22: end for
23: return Res
```

VI. Simulation Results

In this section we report the results of our simulations based on real world dataset. The test scenario contains 100 processing nodes. Network bandwidth of nodes has been chosen randomly from 10 to 100 Mb/s with step of 10 Mb/s. For sampling purpose, we choose 10% of input data and use it to find the distribution of keys in input data.

To evaluate the performance of our proposed algorithm, we have compared it with Hadoop default partitioner and Best performance algorithm. Best performance algorithm knows the distribution of all keys in whole input data (not only sample input data) and also knows the locality of each key in all servers. To know the complete distribution of keys, first we need to process all the input data. It is obvious that it is not efficient to use and implement this algorithm. But this algorithm is a good reference to evaluate other algorithm because it gives the best possible answer. Because our method distributes keys considering their distribution and number of associated values, as well as the processing power, locality and network bandwidth of nodes, it can surpass the default hash partitioner and obtain near Best performance solutions. The results show that our method decreases Reduce tasks' runtime up to 38% compared with Hadoop default partitioner. We have used four well-known applications as our benchmarks;

- WordCount: it counts the number of each key in the file and produces an output file contains all keys and the number of their repetition in input file.
- Inverted Index: this application maps words or numbers, to their locations in a document or a set of documents. This application is used in the most of the text searching systems.
- Grep: an algorithm for finding a key in allover the file. In other words, it finds matching strings from text files and counts their repetition.
- TeraSort: This application is used for sorting files. The goal of this application is to sort a file as fast as possible.

Our data set for this experiment is 10 Gigabyte of English Comments of twitter [15]. This data set contains numerous twitter posts from millions of users all around the world.

Figure 4 presents the total time of Reduce tasks, the Reduce phase execution time, and shuffle execution time of applications under different algorithms. Our proposed method improves the total execution time in Reduce phase compared with default hash partitioner by 32%, 52%, 36%, and 17% for Word Count, TeraSort, Inverted Index, and Grep, respectively. TeraSort has experienced the most improvement while Grep have had the least amount of improvement; because TeraSort produces more intermediate data compared to Grep, and hence, can benefit most from our proposed approach. We can conclude that the higher the volume of intermediate data of an application is, the more it can benefit from our approach. Also, our proposed method improves total execution time of jobs, contains Map and Reduce Time, by 18%, 31%, 14% and 4% for WordCount, TeraSort, InvertedIndex and Grep, respectively.

The different between our proposed algorithm and Best performance algorithm regarding total execution time of Reduce phase is 5%, 8%, 5%, and 0% for Word Count, TeraSort, Inverted Index, and Grep applications respectively. It shows that our algorithm can find near optimal solutions and even optimal solution e.g., in Grep application.

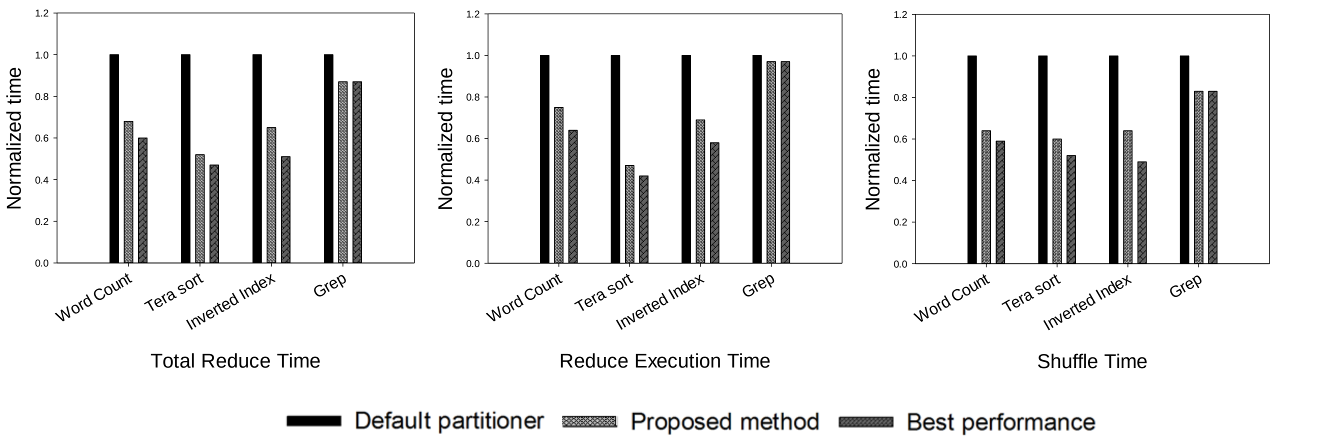

Fig. 4. Normalized finishing time of Reduce tasks for four benchmark (Left: Total Reduce Time, Center: Reduce Execution Time, Right: Shuffle Time)

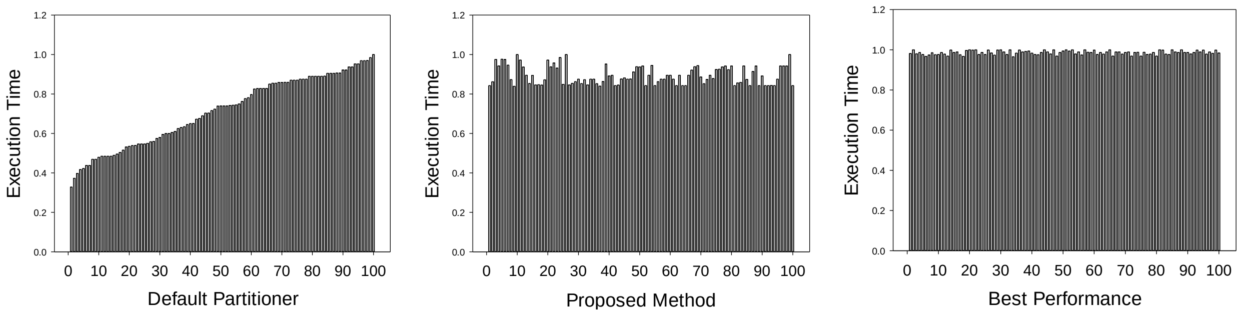

Fig. 5. Normalized finishing time of Reduce tasks when partitioner algorithm is Hadoop default partitioner
(Left: Default partitioner, Center: Proposed Method, Right: Best performance)

Figure 5 depicts the execution time of all the Reduce tasks for WordCount application under different partitioning algorithm. Since the default partitioner is data skew-oblivious, we see significant gap between execution time of different Reduce tasks such that the execution time of the longest task is three times of the shortest one.

On the other hand, Best performance partitioner distributes the keys as efficient as possible between the tasks, and hence, the execution time of all the tasks is almost the same. Please note that as we mentioned earlier, the Best performance algorithm sampling overhead is too much, and hence, its implementation is not efficient and we use it to only show the distance of our approach from best solution.

Finally, our approach stands between default hash partitioner and Best performance, and does it best to balance the execution time of Reduce tasks. The variation of execution time of Reduce tasks in our approach is 5.15 %, while it is 24.76%, and 1.01% for default hash partitioner and Best performance approaches, respectively.

VII. CONCLUSION

In this paper we have proposed a simple and efficient partitioning algorithm to decrease the execution time of Reduce phase in MapReduce applications. Our approach uses sampling to determine the distribution of keys in input data. This approach considers the network bandwidth and processing power of nodes. We have simulated our algorithm in a Hadoop cluster composed of 100 nodes. Results show that our approach can decrease the time of Reduce phase by up to 52% compared with original Hash partitioner of Hadoop and provide results within 8% of the optimal solution.

REFERENCES

[1] Dai, Wei, and Mostafa Bassiouni. “An improved task assignment scheme for Hadoop running in the clouds.” Journal of Cloud Computing: Advances, Systems and Applications 2.1, 2013.

[2] Ranger, Colby, et al. "Evaluating MapReduce for multi-core and multiprocessor systems." IEEE 13th International Symposium on High Performance Computer Architecture, 2007.

[3] http://hadoop.apache.org/

[4] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, “Practical skew handling in parallel joins,” in Proc. of the International Conference on Very Large DataBases (VLDB), 1992.

[5] “HashPartitioner,https://hadoop.apache.org/docs/r1.2.1/api/org/apache/hadoop/Mapred/lib/HashPartitioner.html”.

[6] Chen, Qi, Jinyu Yao, and Zhen Xiao. "*Libra: Lightweight data skew mitigation in MapReduce*." IEEE Transactions on Parallel and Distributed Systems 26.9, 2015.

[7] Ibrahim, Shadi, et al. "*Leen: Locality/fairness-aware key partitioning for MapReduce in the cloud*." Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on. IEEE, 2010.

[8] Hammoud, Mohammad, and Majd F. Sakr. "*Locality-aware Reduce task scheduling for MapReduce*." Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on. IEEE, 2011.

[9] Hanif, Muhammad, and Choonhwa Lee. "*An efficient key partitioning scheme for heterogeneous MapReduce clusters.*" 2016 18th International Conference on Advanced Communication Technology (ICACT). IEEE, 2016.

[10] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, “*Skewtune: Mitigating skew in MapReduce application,*” in Proc. of the ACM SIGMOD International Conference on Management of Data, 2012.

[11] J. Dean and S. Ghemawat, “*MapReduce: simplified data processing on large clusters.*” Commun. ACM, vol. 51, January 2008.

[12] Dean, Jeffrey, and Sanjay Ghemawat. "*MapReduce: a flexible data processing tool*." Communications of the ACM 53.1, 2010.

[13] Seo, Sangwon, et al. "*HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment.*" 2009 IEEE International Conference on Cluster Computing and Workshops. IEEE, 2009.

[14] Yang, Hung-chih, et al. "*MapReduce-merge: simplified relational data processing on large clusters*." Proceedings of the 2007 ACM SIGMOD international conference on Management of data. ACM, 2007

[15] https://snap.stanford.edu/data/twitter7.html