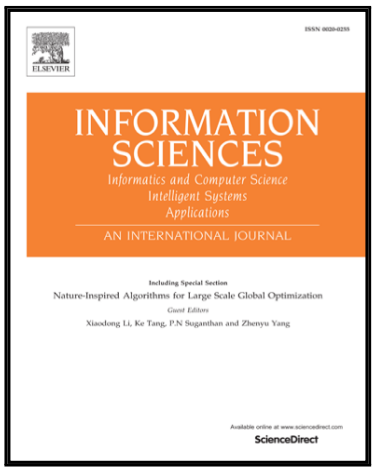

A Novel Algorithm for Handling Reducer Side Data Skew in MapReduce based on a Learning Automata Game

Mohammad Amin Irandoost , Amir Masoud Rahmani , Saeed Setayeshi

PII: S0020-0255(18)30895-8
DOI: https://doi.org/10.1016/j.ins.2018.11.007
Reference: INS 14051

To appear in: *Information Sciences*

Received date: 6 August 2017
Revised date: 27 October 2018
Accepted date: 4 November 2018

Please cite this article as: Mohammad Amin Irandoost , Amir Masoud Rahmani , Saeed Setayeshi , A Novel Algorithm for Handling Reducer Side Data Skew in MapReduce based on a Learning Automata Game, *Information Sciences* (2018), doi: https://doi.org/10.1016/j.ins.2018.11.007

A Novel Algorithm for Handling Reducer Side Data Skew in MapReduce based on a Learning Automata Game

Mohammad Amin Irandoost [1], Amir Masoud Rahmani [1,2,*], Saeed Setayeshi [3]

[1] Department of Computer Engineering, Science and Research Branch, Islamic Azad University, Tehran, Iran

[2] Computer Science, University of Human Development, Sulaymaniyah, Iraq

[3] Department of Medical Radiation Engineering, Amirkabir University of Technology, Tehran, Iran

* Corresponding author

Abstract: In many MapReduce applications, there is an unbalanced distribution of intermediate map-outputs to the reducers. The partitioner determines the load on the reducers. The completion time for a MapReduce job is determined as the slowest reduce task. Under normal conditions assigning a huge amount of data to a task will increase the time required for completion. The current study presents an adaptive algorithm called LAHP (learning automata hash partitioner) that is based on a learning automata game for custom distribution of intermediate key-value pairs to reducers. In this algorithm, a learning automaton on every mapper node is set to control the load on the reducers. This leads to a learning automata game during the execution of a job. This algorithm can partition the intermediate key-value pairs arbitrarily regardless of the statistical distribution of input data and pre-processing. Using the Bonett-test at a confidence level of 95%, the standard deviation ratio of hash-to-LAHP was [0.1, 2858]. This means that LAHP showed much lower dispersion. The results show that the proposed algorithm can successfully distribute any custom load to reducers with an accuracy of over 99% and can speed up the execution of popular applications more than four-fold.

Keywords: Reduce Side Data Skew, Load Balancing, Partitioner, Learning Automata, MapReduce

1. Introduction

The amount of shared data on the Internet is constantly increasing and traditional processing systems are having trouble saving and processing data. A distributed system is required to handle this. The popular term which is used to describe this amount of data is big data [10]. The MapReduce [9] programming model saves and processes such data in distributed systems. The open-source MapReduce programming framework Apache Hadoop [1] is currently in use by Yahoo, Facebook and Google. Data skew often is produced because of the physical properties of objects (e.g. the height of

people distributed normally) and hot spots on subsets of an entire domain (e.g. word frequency in documents following a Zipf distribution) [8].

Data skew can occur in both the map and reduce phases. Because the chunk size is equal, the processing time of map tasks is approximately equivalent. The challenge is for unbalanced loads that are distributed to reducers. The size of the load processed by each reducer is determined by the partitioner function in the map phase. In Hadoop, the default partitioner is a hash function. While the statistical distribution of the keys is uniform, this function distributes the loads to reducers equally. In scientific applications, however, data does not always follow a uniform distribution [8]. Data is skewed in many real-world applications, including scientific applications, database operations such as Join, aggregation functions, search engine functions such as PageRank and invert index and simple applications like sort and grep [29].

When a data skew exists, the load on the reducers becomes imbalanced and the job completion time becomes longer because the slowest reduce task in a MapReduce job determines its finish time. In other words, with the existence of data skew, Hadoop cannot make the best use of the ability of the reducers to reduce. Therefore, handling reducer side data skew is necessary in order to decrease the job execution time and improve system efficiency.

Recent investigations on reducer side skew can be divided into two main categories. One is performing preliminary measures on the whole dataset or a small part of the input and extracting the statistical distribution of data and its frequency to achieve better partitioning for the main job [13, 29, 33, 42]. The other is when the extraction is simultaneously performed in the map phase of the main job. The start time of the reduce phase in this case will be postponed to the end of all or part of the map tasks [8, 11, 14, 18, 27]. The preliminary work for extracting a statistical distribution of data and the lack of parallel execution in the map and reduce phases are considered to be weaknesses in this category, respectively.

The present study introduces a new adaptive partitioning algorithm called learning automata hash partitioner (LAHP) that handles reduce side data skew by adding learning automata to each mapper node. In this algorithm, when the load detected on a reducer is greater than its portion, the learning automaton runs and selects another reducer. With the learning automata, there is no need for preprocessing of data. Furthermore, knowing the statistical distribution of the data in advance is unnecessary. This algorithm has been shown to adapt well to reducer diversity in computational capacity as well as different and a priori unknown user jobs. Also, the load on reducers can be determined during the execution time using a smart scheme. The LAHP map and reduce phases can

be run in parallel. Cluster splitting, which is used in LAHP, can significantly improve load balancing [8, 22, 24, 35, 40]. In summary, the main contributions of this research can be summarized as follows:

- LAHP arbitrarily distributes intermediate keys among reducers regardless of the statistical distribution type without preprocessing, such as sampling of data, postponing shuffle time and reducing the concurrency of the map and reduce phases.

- Heterogeneity is a common issue in data centers as related to the speed of hardware generation over time; however, LAHP has the ability to adapt itself to heterogeneous environments.

- LAHP is implemented in Hadoop. Its performance was evaluated using popular benchmarks and it was compared with state-of-the-art algorithms. The experiment results show that LAHP can improve the job execution time by up to a factor of four in comparison with the default Hadoop partitioner.

The rest of this paper is organized as follows: Section 2 discusses related studies. Section 3 briefly explains the MapReduce technique on Hadoop. Section 4 presents the learning automata. LAHP is proposed for partitioning in Section 5. Section 6 states the reasons that cause a cluster to split and the overhead of the second job is analyzed. Section 7 reports on the results of experiments done to evaluate the performance of LAHP. Section 8 presents the conclusion.

2. Related work

FP-Hadoop, which was introduced by Liroz-Gistau et al. [25], changed the Hadoop internal mechanism by defining a new phase called intermediate reduce with better process map outputs and overcome to data skew. The only difference between the map phase FP-Hadoop and Hadoop are that the map outputs of FP-Hadoop are managed under a set of intermediate reduce fragments used as inputs to the intermediate reduce phase. The algorithm introduced by Chen et al. [8] called LIBRA applies a new sampling method from map output during the mapping process. In LIBRA, in order to achieve suitable precision during partitioning, shuffling should be started after 20% of the map tasks are finished. This algorithm supports total order. However, in LIBRA, the reduce phase is postponed to the end of sampling, which decreases the concurrency of the map and reduce phases.

Gao et al. [13] introduced an algorithm called DLBA that consists of two phases and is based on the greedy algorithm PP. The first phase is a descending arrangement of all map tasks by size. To achieve this purpose, it uses a MapReduce job. In the next phase, based on the arrangement of the previous phase, the task will be assigned to the reducer with the minimum load. SkewTune was introduced by

Kwon et al. [22] to dynamically re-partition unprocessed data of a task if the remaining time is more than one minute and there is an idle node in the cluster. However, because SkewTune does not support cluster splitting, if there is a large cluster in the data, Skewtune cannot handle it and it works only in homogeneous environments.

Lin [24] introduced a theoretical model to show the effect of Zipf distribution in MapReduce performance. Berlińska et al. [6] compared four algorithms for handling reducer side data skew under Zipf in MapReduce. The method introduced by Ramakrishnan et al. [33] estimates reducer load by sampling, splits big clusters and packs medium keys to try to balance the reducer load. Unlike other sampling algorithms that take the required sample size to be stationary, in this algorithm, the user defines a confidence interval. This algorithm requires user knowledge of the statistical distribution or sampling. It is not possible to run the original MapReduce until the sampling phase is complete. Also, this algorithm does not support heterogeneous environments.

A survey of skew in MapReduce applications and mechanisms was carried out by Kwon et al. [21]. The algorithm proposed by Zhang et al. [44] was used to investigate skew caused by heterogeneous machines. It requires a history of job execution time in a heterogeneous environment. Smartjoin, introduced by Slagter et al. [35], considers network traffic when distributing loads on reducers for multiway join. Other researchers [5, 15, 29, 45] have introduced algorithms for handling skew for joins in the MapReduce programming model.

Ibrahim et al. [19] introduced an algorithm called LEEN in which the shuffling start time is postponed to the end of all map tasks to obtain statistics about the keys and their frequency. Then, by using a heuristic method that considers the locality of keys and balancing the reducer load, it can distribute the keys between reducers. LEEN, however, does not support concurrent execution of the map and reduce phases and works only in homogeneous environments. It also assumes that the data size of the key-value pairs is the same.

Gufler et al. [17] developed a method in which the cost of the load that is distributed on reducers is estimated based on a cost model approximation. It uses two histograms for approximations called local and global. Sailfish [34] uses I-files to collect information about intermediate keys based on KFS [3]. It used them to optimize the number of reduce tasks and partition the keys to reducer workers. The straggler problem in a MapReduce job was investigated by Dean et al. [9]. The strategy was for a task to be identified as a speculative when the task progress falls behind the average progress of all tasks at a threshold. This strategy is usually effective for the map task; however, due to its high cost, using it for a reduce task is not effective. Memishi et al. [28] introduced three algorithms for detection and further solving of the straggler problem.

Li et al. [23] investigated a new parallel programming model for handling reducer side data skew which is called Map-Balance-Reduce. In this scheme, when the detected load on a reduce task is over 60%, that reducer stops working and the remaining loads will be distributed fairly to other reducers. However, in this model, there is an overhead to stop the execution of the reducer and to distribute unprocessed data to other reducers. The model does not support heterogeneous environments.

Tang et al. [37] introduced an algorithm called SCID to balance loads on the reducers in the Spark. In this method, a job for sampling must be run first. Then the size of the clusters are estimated based on sampling and the heavy clusters are identified. In the next step, the algorithm attempts to split the heavy clusters so that it can form the balanced load on the reducers. SCID does not support heterogeneous environments and the main job only can be run when the sampling is done and partitioning decision is made.

Liu et al. [26] introduced a method for handling reducer side data skew in Spark streaming. C2WC was introduced by Xu et al. [43]. In this algorithm, before the original job is executed, a MapReduce job is run for sampling. After taking samples and the estimating cluster size, it sorts them in descending order. Then C2WC uses a heuristic method for cluster combinations and assign clusters to reducers. C2WC does not support heterogeneous environments and, when the data skew degree is high, it performs poorly.

Recent approaches require modification in the Hadoop framework. Moreover, a large proportion of them are based on preprocessing and sampling. LAHP, without sampling and Hadoop modification, can effectively distribute the load on the reducers and simply adapt itself to heterogeneous environments. Additionally, LAHP can handle any type of intermediate key skew.

3. MapReduce Programming in Hadoop

The MapReduce project in Hadoop includes the following classes: map, reduce, driver and one optional class called partitioner. Programmers in the map and reduce classes write a specific logic for the map and reduce operations present in the key-value pair model. In the driver class, programmers define the main function and job configuration settings. If the partitioner class is not defined, Hadoop uses a hash partitioner to distribute the intermediate map output to the reducers. Therefore, this function defines the load on the reducers. If reducers are placed on homogeneous machines and this function does not distribute the load on reducers equally, the resulting finish time of the job is determined by the slowest reducer or by the biggest load. The MapReduce is based on the divide and conquer approach [32]. Data processed in this framework is first saved on Hadoop distributed file systems (HDFS) in which data is divided into chunks of the same size that can be deployed on data nodes. Then, map phase begins and mapper nodes run map logic on each chunk [36]. Reduce phase will then start and its results will be saved on the distributed file system [36].

A brief explanation of the work basis of MapReduce on Hadoop 2 is as follows [41]:

1. Job registration by a client;
2. Assigning a unique ID for it and computing the number of input splits;
3. Running YARN (yet another resource scheduler) by the resource manager and assigning and executing a container as AppMaster (application master);
4. A) AppMaster asks for a container for each map task;
   B) Because the number of reducers is determined by the user, AppMaster requests containers equal to that number;
5. Run map tasks and subsequently reduce tasks. By default, whenever 5% of the map tasks are complete Hadoop starts the shuffling process of the reduce phase.

4. Learning Automata

The learning automata [30, 31, 39], as shown in Fig. 1, are machines that deal with the random environment placed in them. A learning automaton has a limited set of actions. It examines the effects of its own previous actions that it randomly selects based on a probability distribution. This selection is made within the stored action-set in the environment and acquires favorability from the circumference response. When arriving at a specific conclusion, it updates the probability of various actions for selecting the best action for different criteria.

The environment is shown as a triple $\{\alpha, \beta, c\}$ in which α is the environment input, β is the output and c is the set of penalty probabilities. The automaton, by performing action α_i with probability c_i receives a penalty from the environment. The environment is divided into the p-model, s-model and Q-model based on output value β. If the environment output is binary $\{0, 1\}$, it is a p-model in which zero and one are desirable actions and undesirable actions, respectively. If environment output is always in the range of [0, 1], it is a s-model. If it is discrete in [0, 1], it is a Q-model.

Learning automata fall into two general families: fixed structure and variable structure. Variable structure learning automata are represented by a triple $\{\beta, \alpha, T\}$ in which β is the set of inputs, α is a set of actions and T is the learning algorithm (a recursive relation) used to modify the action probability vector. Let $\alpha(n)$ denote an action that is chosen at the n moment and $p(n)$ shows the corresponding action probability vector. Let a and b define the reward and penalty parameters, respectively, and let $\alpha_i(n)$ be the action chosen by the automaton at the n moment. When the action taken is rewarded by the environment (β(n) = 0), action probability vector $p(n)$ is updated using recursive Eq. (1), which is a linear learning algorithm.

$$p_j(n+1) = \begin{cases} p_j(n) + a[1 - p_j(n)] & if\ i = j. \\ p_j(n) - ap_j(n) & if\ i \neq j \end{cases} \quad (1)$$

Similarly, in case of penalties, β(n) = 1, the updated formula would be:

$$p_j(n+1) = \begin{cases} p_j(n)(1-b) & if\ i = j. \\ \frac{b}{r-1} + p_j(n)(1-b) & if\ i \neq j \end{cases} \quad (2)$$

In Eq. (2), r is the number of selected actions by the automaton. Based on the values of a and b, three types of learning algorithms have been defined. If the a=b then recurrence Eq. (1) and (2) is called linear reward penalty (L_{R-P}) algorithm, if $a \gg b$ the given equations are called linear reward-εpenalty ($L_{R-\varepsilon P}$) and finally if $b = 0$ they are called linear Reward-Inaction (L_{R-I}). In the latter case, the action probability vectors remain unchanged when the taken action penalized by the environment.

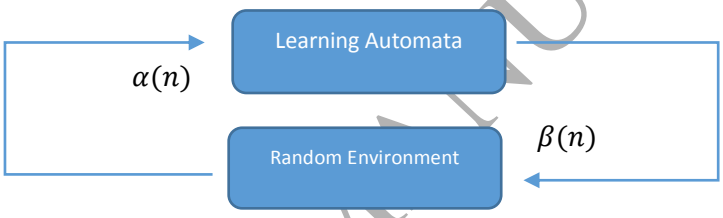

***Fig. 1**. Relationship between learning automata and its environment*

4.1 Variable action-set learning automata

If the number of available actions at each instant changes over time, this learning automaton is called a variable action-set learning automaton (VLA) [38]. In this situation, if A is the set of all actions with n members, then $B(K) \subset A$ is a non-empty subset of actions with m members such that $m \leq n$ (active actions) at time k in which the automaton can select one action randomly according to the scaled probability vector defined in Eq. (3).

$$\begin{cases} \hat{p}_i(k) = \dfrac{p_i(k)}{N(k) = \sum_{i=1}^{m} p_i(k)} & \propto_i \in B(K) \\ \hat{p}_i(k) = 0 & \propto_i \notin B(K) \end{cases} \quad (3)$$

By receiving a reinforcement signal from the environment, the automaton only updates the probability of the active actions. After updating these probabilities, it is necessary to rescale the probability of every action in set $B(k)$ as per Eq. (4).

$$p_i(k+1) = \hat{p}_i(k+1) * N(k) \ \ \forall\, \alpha_i \in B(K) \quad (4)$$

4.2 Game of Learning Automata

If a multi-automata system is viewed as players involved in a game, this configuration is called the game of learning automata [39, 40]. In such game, there are *N* automata that contribute to the game as *N* players. Each of these players independently chooses an action randomly according to its probability. These *N* actions are input into the environment and it responds to them with *N* random payoffs. In this scenario, the probability distribution of the action shows a mixed strategy used by the player at any given instant. By repeating the game, the automata can learn how to solve the game. In this configuration, the reinforcement signal received by each automaton depends on the actions chosen by all automata.

The game can be run in two modes: synchronous and asynchronous. In the first, all players run at the same time and update their strategy at every play of the game. In the second mode, each player can change its strategy at any time or assess the playoff of its current strategy. This absence of synchrony is very common in distributed control systems in which time is continuous and the various elements of the system update their behavior separately [12].

5. Our proposed Algorithm: Learning Automata Hash Partitioner (LAHP)

The proposed adaptive partitioner algorithm is based on an asynchronous game of learning automata. As shown in Figure 2, the map phase in Hadoop has been divided into two subphases: mapping and partitioning.

In the map phase, the mapper nodes begin processing preferably local data and save intermediate results in a circular buffer. When the size of this buffer reaches the threshold, the spill buffer to the local disk will start. Before spilling, the partition function calls to allocate key-value pairs to reducers for processing. The mapping operation is always prior to partitioning. The LAHP was designed based on this order. The notations used in this paper are summarized in Table 1.

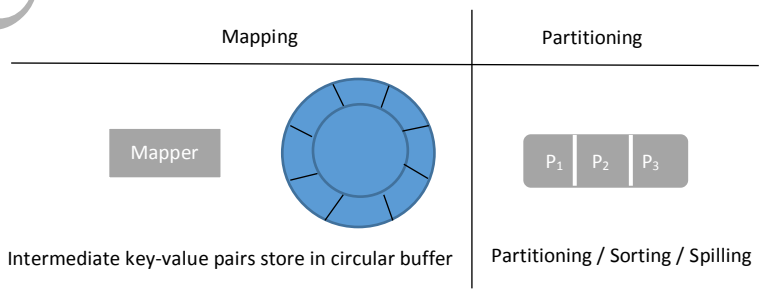

***Fig. 2**. Map phase includes two sub phases: Mapping and partitioning*

Table 1: *The notation that used in this paper*

Name	Description
L_i	The Load placed on reducer i.
KC	Key Counter.
A	The number of Active reducers (actions, in VLA).
H	Reducer selected by Hash partitioner
VLA	Variable action-set Learning Automata.
TL	Total present Loads distributed to all reducers
F	Frequent Interval.
RP	The Reducer portion from distributed loads.
ALRTP	Average ratio of active reducer load to the portion already distributed.
RC_i	The Capability of Reducer *i*.
MAPE	Mean Absolute Percentage Error.

5.1 Proposed Partitioning Algorithm

To effectively partition a data set consisting of *K* keys on *N* reducers, the best solution in a space of N^K possible solutions must be found [19]. Given that all possible solutions are too many to explore, LAHP uses an automata–based approximation algorithm to solve the reducer side data skew problem. In LAHP, when data skew is detected during a job, a game of learning automata is used to distribute the load to the reducers. Each mapper has a learning automaton and contributes to the game as a player. Every player, based on the probability vector, chooses an action independently. Depending on the environment response of reward or penalty for the actions chosen by other automata, the player updates its probability vector. By repeating the game, each player learns to choose the optimal action from among all actions. The LAHP process is as follows:

1- Each mapper has a learning automaton. In all automata, we define action-set as $\{\alpha_1, \alpha_2, \dots, \alpha_r\}$, in which r is equal to the number of reducers. The action-probability vector of the learning automaton is specified as $p(n) = \{p_1, p_2, \dots, p_r\}\ at\ instance\ n$ and initializes them to $1/r$. Let the reducer capability vector be $RC = \{RC_1, RC_2, \dots, RC_r\}$. If all reducers are homogeneous machines**,** the capability values are assumed to be equal for all reducers**.** However, if the reducer machines are heterogeneous, the values depend on the capability of each machine. For example, if the load on machine A should be as twice as much on machine B, it is sufficient to set $\mathrm{RC_A} = \mathrm{RC_B} \times 2$.

2- In this algorithm, we use r global counters $\{L_1, L_2, \dots, L_r\}$, in which r is the number of the reducers. These counters save the cost of the load placed on each reducer in order to acquire the amount of each reducer's contribution based on LAHP. The initial value of these counters is zero.

3- In the map or combiner function, for each output key, the relative reducer is determined by the array x=*MapHash2LAHP* [HashPartitioner[1](key)] that will be described in step 4a, and its counter (L_x) will be added by the return value of the cost function (key, value, *m*, *x*) where *m* is the mapper number and *x* is the reducer number. In general, this function can compute various costs, such as those for communication and computation according to the application.

4- The partitioning phase contains the following steps:

a. For mapping, the reducer selected with a hash partitioner to LAHP considers local array *MapHash2LAHP= {0, 1,…,r-1}* by size *r* in which *r* is equal to the number of reducers in each mapper. For job initialization, LAHP and the hash partitioner work similarly. However, during job progress, if the observed load cost of the reducer selected by the hash is greater than that of its portion (data skew), the learning automaton will run and select a reducer from among the other reducers that has less load than of its portions. i. j are reducers selected by hash and the LAHP partitioner, respectively.

$$MapHash2LAHP[i] = \begin{cases} i & time = 0 \\ j & time > 0 \end{cases} \quad (5)$$

b. Consider frequent intervals (*F*) for executing learning automata. These intervals have a direct relation to the job size. If the job is large, a larger period should be chosen and vice versa. In fact, this parameter determines the interval for choosing optimal actions in the automata to overcome data skew. When this interval is very small, other automata (players) do not have sufficient opportunity to respond to the strategy selected in the previous step. For large intervals, the automata do not have sufficient opportunity to choose an optimal action before the end of a job. Consequently, the load distribution will become unbalanced.

c. Consider a local counter such as *KC* with an initial value of zero. This value increases by one when a key enters the partitioner.

d. For an incoming key entering at a frequent interval (*F*), suppose *H* is the reducer selected by the hash partitioner for the key.

e. The total load cost distributed to all reducers up to now must be calculated using to Eq. (6).

$$TL = \sum_{i=1}^{r} L_i \quad (6)$$

[1] (key.hashcode() & integer.maxvalue)%the the number of reducers

f. If L_H (load cost on reducer *H up to now)* is neither more than nor equal to its portion (*RP(H)*) × (1+threshold), set *MapHash2LAHP[*HashPartitioner*(key)]=H* and return *H*. *RP(H)* is computed using Eq. (7) and the threshold is the maximum percentage at which the load can be tolerated, otherwise data skew occurs and the learning automaton starts to select a reducer randomly according to probability. The following cases can occur:

$$RP(n) = TL \times RC_n / \sum_{j=1}^{r} RC_j \quad (7)$$

i. In the action set of this automaton, all reducers (such as *H*) that have a load cost greater than that of their portion $RP(H) \times (1 + threshold)$ must be removed, as described in Sect. 4.1 on VLA. This increases the convergence rate and convergence speed of the automaton. Suppose the number of active reducers is equal to *A*.

ii. The automaton selects a reducer such as *i* according to its (scaled) probability vector for all active actions (reducers).

iii. Calculate the average dynamic load cost ratio of the active reducers according to their portions that have already been distributed and save it to $ALRTP$ as computed using Eq. (8).

$$ALRTP = \frac{\sum_{k=1}^{A} \frac{L_k}{RP(k)}}{A} \quad (8)$$

iv. Because the most important purpose of handling reducer side data skew is for all reducers to finish their works almost simultaneously, if the ratio of reducer load cost to reducer portion $\frac{L_i}{RP(i)}$ is lower than $ALRTP$, action i gives a reward ($\beta = 0$) and otherwise it gives a penalty ($\beta = 1$).

v. The reducer election probability should be updated using Eqs. (1) and (2).

vi. Array *MapHash2LAHP* should be updated according to the reducer selected in step ii as follows:

*MapHash2LAHP[*HashPartitioner*(key)]=i.*

This means that the load on the reducer selected by the hash partitioner is placed on the reducer which is selected by LAHP. This continues at least until the next frequent interval.

vii. The learning automaton again enables the removed actions according to Eq. (4).

viii. Return R_i.

g. Outside the frequent interval, operate as follows:

if $L_H < (1 + threshold) \times RP(H)$

*MapHash2LAHP[*HashPartitioner*(key)]=H.*

Return *H*.

Else

Reducer = MapHash2LAHP[hashPartitoner(key)] is selected and returned.

5.2 Pseudo Code for LAHP in MapReduce Applications

The pseudo code for LAHP partitioner is presented below.

Algorithm 1. LAHP partitioner.

Inputs:

k: The key of key/value pairs for specifying reducer

r: The number of reducers

Output:

Reducer number

Assumptions

1: Initialize r-dimensional action-set: $\alpha = \{\alpha_1, \alpha_2, \dots, \alpha_r\}$ with *r* actions.

2: Initialize r-dimensional action-probability vector: $p(n) = \{p_1, p_2, \dots, p_r\} = \{\frac{1}{r}, \frac{1}{r}, \dots, \frac{1}{r}\}$ at instant *n*.

3: Initialize r-dimensional vector $RC = \{RC_1, RC_2, \dots, RC_r\}$. This vector defines the capability of each reducer, in homogeneous machines, it is assumed that all values are equal.

4: Initialize *F* as a frequent interval.

5: Initialize r-dimensional *MapHash2LAHP*={0,1,…,r-1}.

6: Set *r* global counter {L_1, L_2, …, L_n} with zero initial value.

7: In the map or combiner function for each output key, the relative reducer is determined according to *x=MapHash2LAHP[HashPartitioner (k)]* and its counter added by the return value of cost (key, value, m, x) function.

8: Define a local counter by name such as *KC* with zero initial value.

Begin

1: increment *KC*.

2: **If *KC* % *F* == 0 then**

3: Set *H*= HashPartitioner(k)

4: Calculate the total load cost as the sum of loads cost that placed on all reducers until now and save it into *TL* according to Eq. (6).

5: ***if*** $L_H \geq (1 + threshold) \times RP(H)$ ***then***

6: In automaton action-set, all reducers such as *H* that have $L_H \geq (1 + threshold) \times RP(H)$ must be Removed and probability of remaining actions should be updated according to VLA Eq. (3).

7: Set *A*= the number of remaining actions (active action).

8: Automaton select an action or another word selects a reducer such as R_i randomly according to its (scaled) probability vector of all active actions.

9: Compute $ALRTP$ according to Eq. (8).

10: ***if*** $\frac{L_i}{RP(i)} < ALRTP$ ***then***

11: Beta=0

12: The selected action by learning automaton is rewarded according to Eq. (1).

13: **Else**

14: Beta=1

15: The selected action by learning automaton is penalized according to Eq. (2).

16: **End If**

17: MapHash2LAHP[hashpartitioner(k)] = R_i.

18: Learning automaton enables the removed actions again according to Eq. (4).

19: **Return R_i.**

20: **Else**

21: MapHash2LAHP[hashpartitioner(k)] = H.

22: **Return *H*.**

23: **End If**

24: **Else**

25: ***if*** $L_H < RP(H) \times (1 + threshold)$ ***then***

26: MapHash2LAHP[hashpartitioner(k)] = H.

27: **Return *H***

28: **Else**

29: R_i = MapHash2LAHP[hashpartitioner(k)].

30: **Return R_i**

31: **End If**

32: **End If**

End

5.3 Time complexity analysis

In this subsection, the time complexity of LAHP is analyzed by closely following the procedure for selection of a reducer for each key-value pair as received from map output. LAHP is separated in the following steps:

1. In the frequent interval (lines 3 - 24), the following occur:
   i. Determine the reducer selected by the hash partitioner for the key received: O(1).
   ii. Calculate the total load cost: O(r).
   iii. Compare the load on the reducer selected in the step 1(i) by its portion: O(r).
   iv. If the result of the comparison in step 1(iii) is true, disable some actions: O(r).
   v. Update the probability vector of active actions: O(A).
   vi. Select an action: O(A).
   vii. Compute $ALRTP$: O(A).
   viii. Reward or penalize the selected action: O(A).
   ix. Enable the disabled actions: O(r).
   x. If the result of the comparison in step 1(iii) is false, update the array and return the selected reducer: O(1).
2. Outside of the frequent interval (lines 25-32), LAHP computes the portion of reducer selected by the hash partitioner, compares it with its load and updates an array: O(r).

   The time complexity of LAHP is $O(r) = Max\{\,Time\ complexities\ of\ steps\ 1\ and\ 2\}$; therefore, the time complexity of LAHP for distributing *m* key-value pairs is $O(\,mr)$.

   The time complexity of SCID [37] for cluster combination and splitting is $O(rn^2)$ and this algorithm has time complexity $O(Max\{n,r\})$ for dispatching a cluster for each key-value pair. Therefore, the time complexity of SCID for distributing *m* key-value pairs is $O(Max\{rn^2\,, mn, mr\})$ where *r* is the number of reducers and *n* is the number of clusters. The time complexity of C2WC [43] in the cluster combination by assuming use of a sorting algorithm with complexity $nlogn$ is $O(Max\{nlogn, nrlogr\})$ and in cluster dispatching using a linear search algorithm for each key-value pair is $O(n)$; therefore, the time complexity C2WC for distributing *m* key-value pairs is $O(Max\{nlogn, nrlogr\,, mn\})$. SCID and C2WC are based on sampling which requires a separate job to be run before the main job for the purpose of obtaining a variety of clusters and their sizes. LAHP can handle data skew in MapReduce without postponing shuffle time or requiring a preprocessing step such as sampling of data.

6. Discussion about Cluster Split in LAHP and Integrative Job

All keys with the same value are called a cluster. In the hash partitioner, all members of one cluster are sent to one reducer only. This is not necessary for the following reasons.

6.1 Cluster Split

a. In applications such as sort, grep and total join, if all members of one cluster do not send data to one reducer only, there will be no disturbance in the final result [8]. In other words, mapper output can be sent to any reducer in this application [8].

b. If members of some clusters are further away than others and those clusters map to only a small percentage of reducers, it will result in more load on those reducers compared to the others. Consequently, the slowest reducer will determine the job finish time. For example, suppose that the intermediate map output word count application includes three clusters having 1000, 50 and 50 members. If we consider two reducers, the best load distribution that can be done without cluster splitting by partitioners is as follows:

   $Reducer_1$: 1000 keys
   $Reducer_2$: 50+50 keys

   As a result, the load on $reducer_1$ is 10 times than that placed on $reducer_2$.

Gray et al. [16] classified aggregation functions to three groups: distributive, algebraic and holistic. Liroz-Gistau et al. [25] used this classification on reduce functions. For all distributive and algebraic reduce functions, LAHP can be used as a partitioner. It also can be used for some holistic reduce functions. LAHP always considers a cluster split. After the main job is finished using LAHP, an integrative job with a hash partitioner must be run in order to integrate the results. This does not apply to applications that do not need to send all values of a cluster to one reducer only. The output of the main job is considered to be integrative job input. Mathematical analysis and implementation results show that the time required to run the second job is too short; thus, the execution time of both jobs (main and integrative job) is much less than one main job with a hash partitioner.

For example, suppose that the intermediate map outputs of a job contain three unique clusters as described in section 5(1b). If the loads of the three clusters are distributed to two reducers fairly, in the optimal case, the load on each reducer will be 550 keys. However, if the partitioner does not consider a cluster split, the best load distribution which can be done is 1000 keys for $reducer_1$ and 100 keys for $reducer_2$. Therefore, in comparison with the optimal distribution, $reducer_1$ must process an additional 450 keys. Although LAHP, in applications which must send all of the same keys to one reducer, must execute an additional job for integrating of the results, it consumes much less time in comparison with having the same main job with a hash partitioner. In this example, there are three unique keys A, B, C in which B and C are placed on $reducer_2$. In the worst case in LAHP, at the end of the reduce phase, each reducer has one sample of each unique key. Therefore, the integrative job processing overhead on the reduce phase is max {two keys on $reducer_1$, four keys on $reducer_2$}. LAHP will totally process 550+4 keys. As a result, the load decreases at least 44.6% on $reducer_1$. This example is shown in Table 2.

6.2 Integration Methods

In order to integrate the results produced by LAHP, two methods can be considered. In the first method, without changing the Hadoop internal mechanism, it uses an extra job according to the hash partitioner which it applies to the output of LAHP for integrating the results of job_1. In the second, an internal Hadoop mechanism is changed by defining two reducer functions. The first and second reducer functions are based on LAHP and the hash partitioner, respectively. In this method, after completion of all reduce tasks, the output of the first reduce function by the hash partitioner is sent to the appropriate reducer and second reduce function is run to integrate the results. In the current study, the first method was used.

Table 2. *Job_1 uses a partitioner with two reducers that does not do cluster split. Job_2 uses LAHP with two reducers. Job_3 is an integrative job, the input of which is the output of job_2 and which the Hash partitioner distributes to two reducers. Note: cnt is an abbreviation for "count".*

Job #	Intermediate map outputs	Reducer input	Job output
Job_1	A, cnt=1000 B, cnt=50 C, cnt=50	(Exist data skew on reducers) $Reducer_1$: A, cnt=1000 $Reducer_2$: B, cnt=50 C, cnt=50	$Reducer_1$: A, value $Reducer_2$: B, value C, value
Job_2	A, cnt=1000 B, cnt=50 C, cnt=50	(Does not exist data skew) A, cnt=500 B, cnt=25 C, cnt=25 A, cnt=500 B, cnt=25 C, cnt=25	A, value B, value C, value A, value B, value C, value
Job_3 (Integrative)	A, cnt=2 B, cnt=2 C, cnt=2	A, cnt=2 B, cnt=2 C, cnt=2	A, value B, value C, value

6.3 Integration Job

This job must be defined according to the application. Here we introduce some applications and their integrative jobs.

6.3.1 Count or Sum Application

The integrative job for count or sum application.

```
1:  Map(k,v)
2:      For each key in content do
            Emit(k,v)
```

```
3:      End
4:  Reduce(k,list v)
5:      S=0
6:      For each member in list v such as u do
            S+=u.value
7:      End
8:      Emit (k,s)
```

6.3.2 Min or Max Application

The integrative job for min or max application.

```
1:  Map(k,v)
2:      For each key in content do
3:          Emit(k,v)
4:      End
5:  Reduce(k,list v)
6:      In list v find min or max and save it in min or max
7:      Emit (k, min or max)
```

6.3.3 Avg Application

The integrative job for avg application.

```
1:  Map(k,v)
2:      For each key in content do
3:          Emit(k,v)
4:      End
5:  Reduce(k,mixed list v include sum and count)
6:      For each member in list v such as u do
            S+=u.sum
            count+=u.count
7:      End
8:      Emit (k,s/count)
```

6.4 Integrative Job Complexity Analysis

The complexity of the integrative job is analyzed below.

Theorem 1. In the worst case, the overhead of the integrative job is equal to the required processing cost of the unique keys of the main job.

Proof. Suppose that the total number of keys which must be processed by the main job is equal to n and the number of unique keys in this job is m such that $m<<n$ and r is the number of reducers. Because the number of distinct keys is m, after executing the main job, there will exist in each reducer one delegate for each key. In the worst case, LAHP produces m unique keys in every reducer. Consequently, the total number of keys that an integrative job must process in the worst case is equal to $r \times m$. Using a hash partitioner, the contribution of each reducer is approximately equal to m keys. The cost required for processing $r \times m$ keys is equal to the approximate processing cost of m keys.

7. Performance Evaluation

LAHP was implemented on Hadoop 2.7.1. The experimental setup, including the experimental platform, datasets and queries are first discussed. Next, the results of tests done to study LAHP performance in different situations is discussed.

7.1 Setup

Experiments were run on a cluster with seven virtual machines on an HP Proliant DL580G7 server (KVM as the hypervisor) [20]. Table 3 shows the properties of the machines.

Table 3. *The properties of seven machines.*

Type	#	CPU	Ram	HDD	OS
Master	1	Intel® Xeon CPU E7-4870 30 MB SmartCache 4 CPU – 2 core per CPU	12 GB	30GB	Ubuntu 16.04 LTS
Worker	6	//	//	//	//

All experiments used the default configuration in the Hadoop. Linear learning scheme L_{R-P} with equal reward and penalty parameters values of 0.015 were used in all experiments. Because the implementation platform was limited and the hard disks of the machines had limited space, no combiner function was used in the experiments, which allowed a better examination of the performance of LAHP.

It is assumed that the return value of the cost function for each key-value pair is 1 and the threshold is set to 1%. Synthetic and real datasets were used to evaluate LAHP and three benchmarks were used: word count, grep and inverted index. The synthetic dataset was generated using a zeta distribution (Zipf) with the value of σ being 0.5 to 5.0. The real dataset was downloaded from [2]. All experiments were conducted five times and the average results are reported. The probability density function for the zeta distribution is:

$$f(x) = \frac{1}{x^{\sigma} \sum_{i=1}^{n} (1/i)^{\sigma}} \quad for\ all\ positive\ integers\ n\ and\ all\ \sigma > 0 \tag{9}$$

LAHP was compared with the following algorithms:

- Hash: a default partitioner used in Hadoop and Spark for partitioning [4].
- SkewTune: During the job execution time, it detects a straggler node and distributes the remaining loads among other nodes [22].
- SCID: Before the main job, a sampling job is run. After estimating a variety of clusters and sizes, SCID try to overcome data skew by cluster combination and cluster splitting [37].
- C2WC: Uses a heuristic method for cluster combination after sampling by a separate MapReduce job [43].

7.2 Criteria of Evaluation

To compare LAHP with the algorithms in subsection 7.1, the following criteria were used:

- Job runtime: This parameter is equal to the job finish time – job start time.
- Reduce time: This parameter is equal to reduce finish time – shuffle start time.
- Percentage of load on each reducer: To compute this parameter it is sufficient to calculate $\frac{Reducer\ Load}{Total\ Load}$ and multiple it by 100 .
- Coefficient of variation of the distributed load (COV): COV is a common measurement for data skew, which is computed as $\frac{STDdev(\overrightarrow{load})}{avg(\overrightarrow{load})}$, where $STDdev$ is standard deviation and load is a vector that includes the load placed on each reducer.

7.3 Experimental Results

This section is divided into two parts. Part one reports on the results on the synthetic dataset. In part two the results for the real dataset are shown.

7.3.1 Experiments on Synthetic Dataset

In all experiments, the 4.2-GB synthetic dataset produced by a Zipf distribution was used.

Experiment 1: In this experiment, $\sigma = 5.0$ was set for generating the dataset. The number of reducers changed from 2 to 12 and the word count benchmark was run by LAHP and other algorithms. Fig. 3 shows that the job execution time by LAHP outperformed the others at any number of reducers. When extreme skew existed in the input data, Hash, SkewTune and C2WC performed badly because an increasing the number of reducers did not have a considerable effect on job execution time. The execution time decreased substantially for LAHP and SCID by increasing the number of reducers. Because C2WC is based on sampling and cluster combination, when the data skew was high, C2WC could not handle it using cluster combination. In this algorithm, the startup of the main job was

postponed to the completion of sampling and partition decision making. For this reason, the job execution time by C2WC was higher than for Hash at a higher degree of skew. SkewTune has extra overhead, such as resource competition, and does not support cluster splitting.

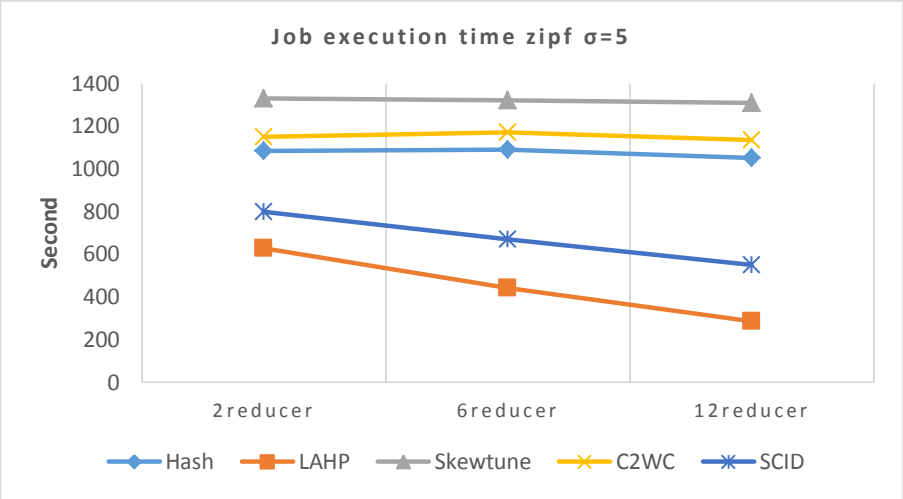

***Fig. 3**. Varying reducer number*

Fig. 4 shows the elapsed time for 12 reducers. In Hash, SkewTune and C2WC, one reducer took a much longer time because of data skew, whereas, in LAHP, all reducers took approximately the same amount of time. Because SCID is also based on cluster splitting, it worked better than Hash, C2WC and SkewTune; however, the elapsed reducer time shows variations because the loads on the reducers were not equal. The loads placed on 12 and 6 reducers are shown in Figs. 5(a) and 5(b), respectively. Clearly, using LAHP, the load percentage of each reducer was approximately $\frac{100}{12}$ and $\frac{100}{6}$, respectively, and there was no data skew on the reducers. In Hash, C2WC and SkewTune, almost 96% of the load was placed on reducer$_1$.

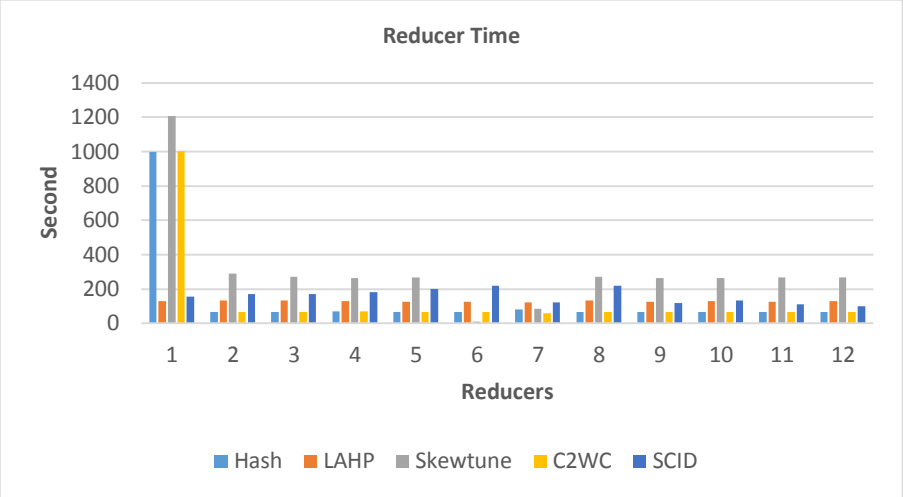

***Fig. 4**. Elapsed reducer time*

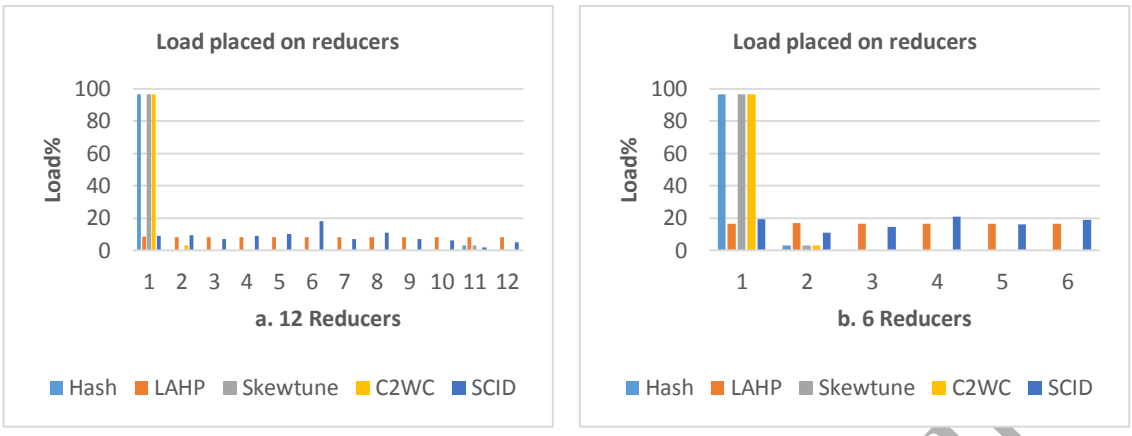

Fig. 5 *a. Percentage of load placed on 12 reducers; b. Percentage of load placed on 6 reducers.*

The COV of the loads placed on reducers by different algorithms is shown in Fig. 6. This parameter measures the ratio of standard deviation to mean. A larger coefficient indicates heavier skew. As shown, this ratio for LAHP using 12 reducers was only 0.005, while the COV for Hash was 3.1893. The optimal value of COV in homogenous environments is zero, which indicates that the load has been distributed among reducers quite fairly. After LAHP, SCID performed better than the others because this algorithm is also supported cluster splitting.

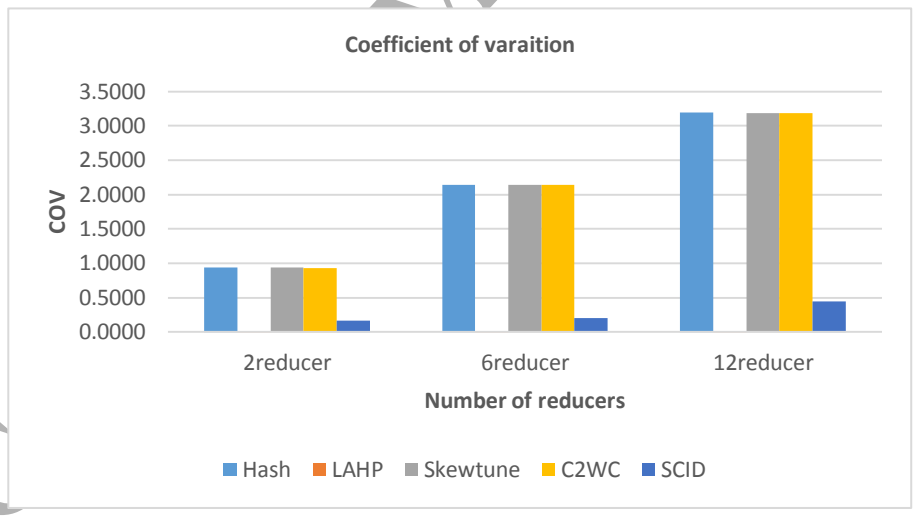

Fig. 6*. Coefficient of variation versus number of reducers*

Experiment 2: In this experiment, to evaluate the performance of LAHP for different amounts of skew, a 4.2 GB synthetic dataset with a Zipf distribution was used and the values of σ from 0.5 to 5.0 controlled the degree of skew. A larger value of σ means a heavier skew. In this test, the number of reducers was set to 12. The job execution time is shown in Fig. 7. It is clear that skew had a significant impact on hash, C2WC and SkewTune, whereas LAHP was skew-protected and performed better than the others. The job execution time for LAHP at $\sigma = 0.5$ (light skew) was slightly more

than for Hash, SkewTune and C2WC because it required additional work to select the best reducer and integrative job. At a heavier skew, the job execution time by LAHP was approximately 25% that of Hash ($SpeedUP\frac{Hash}{LAHP} \cong 4$).

***Fig. 7**. Job execution time versus σ*

The COV for LAHP and the other algorithms at different values of σ are shown in Fig. 8. LAHP, for all degrees of skew had a near-zero COV (0.003, 0.006), whereas, in Hash, C2WC and SkewTune, as the degree of skew increased, the COV grew rapidly. The COV of SCID was lower than for Hash, C2WC and SkewTune; however, LAHP had the lowest COV.

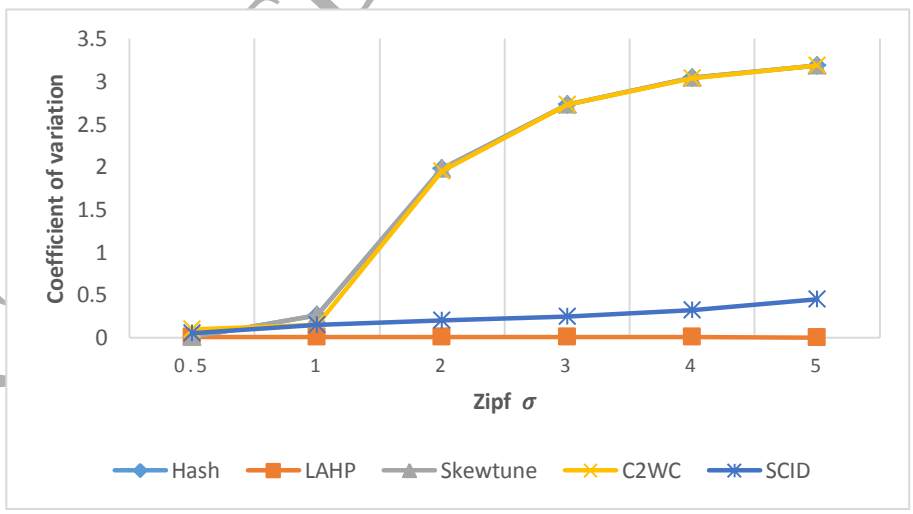

***Fig. 8**. Coefficient of variation versus σ*

Experiment 3: In the heterogeneity test, σ = 0.5 (light skew) because the goal of this test was to evaluate LAHP for placement of various amount of loads on reducers. Suppose six reducers with 40%, 7%, 13%, 20%, 15% and 5% of loading are expected, the experimental results are depicted in Fig. 9. LAHP had the best results for this test; demonstrating that it is a flexible algorithm which can

work well in a heterogeneous environment. The MAPE and accuracy were calculated according to according to Eqs. (10) and (11), respectively. EL_i is the percentage of expected load and AL_i is the percentage of actual load on reducer i.

$$MAPE = \frac{\sum_{i=1}^{r} \frac{|EL_i - AL_i| * 100}{EL_i}}{r} \tag{10}$$

$$Accuracy\ percentage = 100 - MAPE \tag{11}$$

In this Experiment MAPE and accuracy are computed as follow:

$$MAPE = 0.575\%$$

$$Accuracy = 99.425\%$$

The experiment was repeated 10 times with different expected loads for each reducer and an average accuracy of 99.1% was obtained.

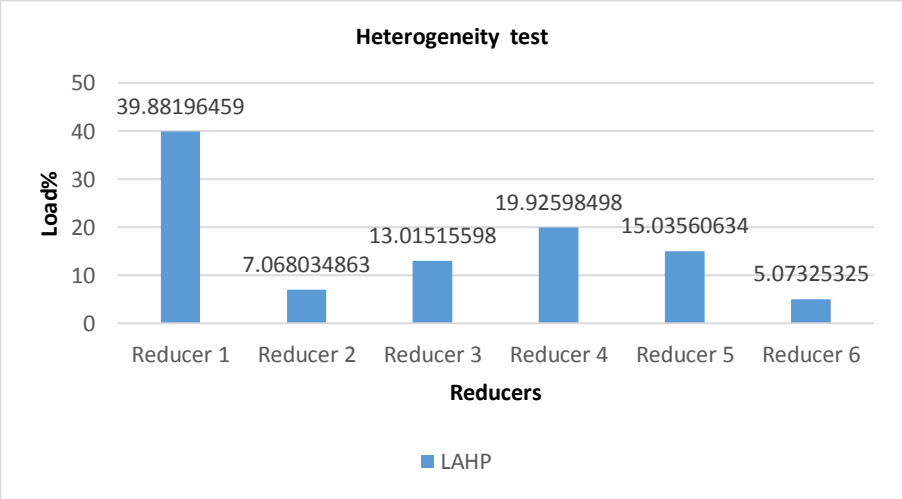

***Fig. 9**. Heterogeneity load on six reducers*

Experiment 4: In this experiment, we ran the word count benchmark using LAHP and the Hash partitioner in different situations by varying σ and the number of reducers and by computing the COV. The results are shown in Table 4. The table shows that, at all values of σ with any number of reducer, the LAHP COV was significantly lower than the Hash COV. At a heavier skew of σ = 5 using 12 reducers, the LAHP COV was only 0.00253 while the Hash COV was 3.331. At every σ, the Hash COV was directly related to the number of reducers whereas LAHP COV was approximately fixed for all variations.

Table 4. *Hash and LAHP COV on different σ and number of reducers*

σ	2 reducers		6 reducers		12 reducers	
	Hash	LAHP	Hash	LAHP	Hash	LAHP
0.5	0.001138	0.00006	0.002797	0.000908	0.004800	0.004920000
1	0.079305	0.00002	0.173382	0.000657	0.275201	0.003822279
2	0.707142	0.00003	1.397555	0.000952	2.071223	0.003902856
3	1.060683	0.00005	1.977513	0.000749	2.852716	0.004215653
4	1.237429	0.00002	2.230974	0.001159	3.182849	0.004318502
5	1.325852	0.00001	2.346051	0.001183	3.331158	0.002535249

Experiment 5: The value of σ was set at 5 and a 4.7-GB synthetic data set was generated. By varying the number of reducers, the performance of the word count, inverted index and grep benchmarks were evaluated for the Hash partitioner and LAHP. The results of this test are shown in Fig. 10. Clearly, for all three applications and every number of reducer, the job finish time for LAHP was much lower than for Hash. Furthermore, increasing the number of reducers decreased the job finish time of the LAHP, whereas the job finish time for Hash showed no appreciable change.

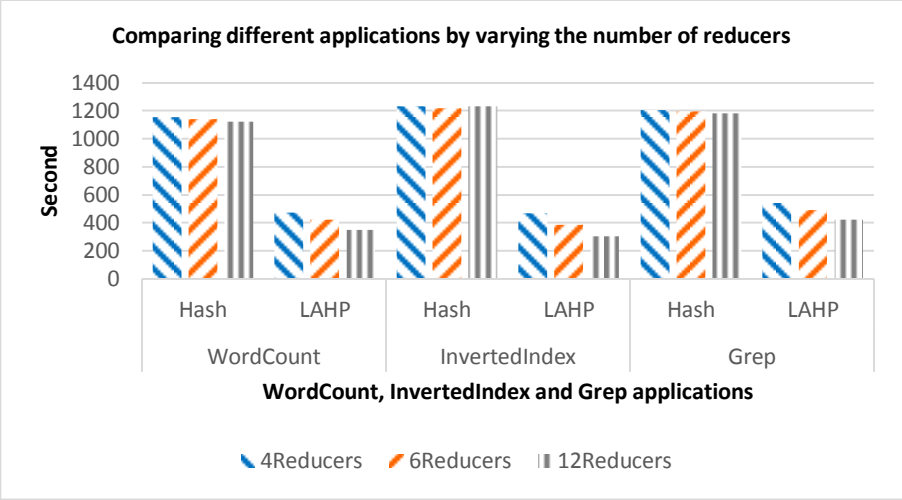

Fig. 10*. Comparison of applications versus the number of reducers*

7.3.2 Experiments on Real Dataset

We evaluated the use of LAHP on a 4.1 GB real dataset [2].

Experiment 6. The word count benchmark was run for LAHP and the Hash partitioner. The number of reducers was set at 4. As shown in Fig. 11, the load distribution by LAHP for each reducer was very close to optimal. The job execution time and COV of this experiment are shown in Table 5. It is clear that the job execution time and COV for LAHP were lower than for Hash.

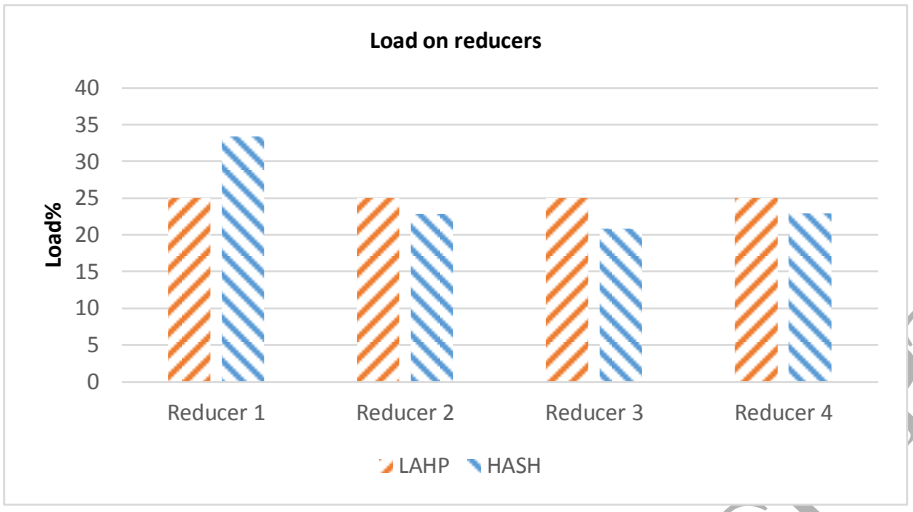

***Fig. 11**. Load on four reducers*

***Table 5**. Experiment results on real data set*

	Hash	LAHP
Job execution time (second)	467.4	410.75
Coefficient of variation	0.226530214	0.000639346

7.3.3 The Bonett-test

The statistical Bonett test [7] is used if the standard deviation of the two populations are equal and the ratio of standard deviations should be estimated with the desired confidence and precision. In this test, the null hypothesis was σ (Hash load)/σ (LAHP load) = 1 and the alternative hypothesis was σ (Hash load)/σ (LAHP load) $\neq$ 1. From the results given in Table 6, for $\sigma = 0.5$, the null hypothesis was accepted because p = 0.942 $>$ α = 0.05. As a result, Hash StDev (standard deviation) and LAHP StDev are approximately equal. For the other σ, because p = 0.000 $<$ α = 0.05, the alternative hypothesis was accepted. For example, at σ =5.0, the Hash StDev was 27.76% and LAHP StDev was only 0.021%. Therefore, the ratio of Hash StDev to LAHP StDev was 1313.937. At a confidence level of 95%, the StDev of the distributed load percentage by Hash and LAHP were [8.228%, 111.934%] and [0.015%, 0.037%], respectively. Based on the Bonett test at a confidence level of 95%, the StDev ratio of Hash partitioner to LAHP was at least 147.727 and at most 2857.975. This means that the dispersal by LAHP was much lower than for Hash.

Table 6. *The Bonett test results*

			Ratio of	95% CI for		95% CI for		95% CI for StDev		
	StDev%	StDev%	standard	StDev% Hash		StDev% LAHP		Ratio(Bonett)		p-value
σ	Hash	LAHP	deviations	Min%	Max%	Min%	Max%	Min	Max	(Bonett)
0.5	0.003999	0.040983	0.976000	0.022	0.08800	0.031	0.066	0.10700	3.111000	0.942
1	2.292952	0.031847	71.99900	0.916	6.86400	0.020	0.062	13.6720	177.2550	0.000
2	17.25976	0.032523	530.6940	5.379	66.1980	0.020	0.062	65.0710	1288.845	0.000
3	23.77233	0.035130	676.6960	7.056	95.7330	0.024	0.061	76.2530	1475.960	0.000
4	26.52336	0.035987	737.0260	7.839	107.258	0.027	0.057	82.4560	1477.218	0.000
5	27.75823	0.021126	1313.937	8.228	111.934	0.015	0.037	147.727	2857.975	0.000

8. Conclusion

The current study developed a partitioner called LAHP to handle reducer side data skew in MapReduce. LAHP, which is based on a learning automata game, can handle data skew very efficiently. This algorithm can manage any type of intermediate map output skew and does not require sampling. It can also adapt very well to heterogeneous systems. The performance of LAHP was evaluated in experiments using synthetic and real datasets. The results show that LAHP, with minimum overhead, can distribute the load more equitably to the reducers than other state-of-the-art algorithms. The results of the experiments reached better than 99% accuracy for load balancing. The results of the statistical Bonett test [7] on LAHP and Hash partitioner showed that, at a confidence level of 95%, the standard deviation ratio of Hash-to-LAHP was [0.1, 2858]. This means that the standard deviation ratio of the distributed load by LAHP partitioner has less dispersion around the mean compared to that of the Hash partitioner. LAHP can also handle data skew caused by size. This means that data skew could also occur when the sizes of the keys are different and affect the shuffle time. In future works, this issue will be considered. Furthermore, the effectiveness of LAHP in the presence of the combiner operation will be investigated and the communication cost among mappers and reducers will be considered.

References

[1] Apache Hadoop, in, http://hadoop.apache.org/.

[2] Billion Word Imputation, in, https://www.kaggle.com/c/billion-word-imputation/data.

[3] KFS, in, https://code.google.com/p/kosmosfs/.

[4] Hash Partitioner, in, https://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/mapreduce/lib/partition/HashPartitioner.html.

[5] F.N. Afrati, N. Stasinopoulos, J.D. Ullman, A. Vassilakopoulos, SharesSkew: An algorithm to handle skew for joins in MapReduce, Information Systems, 77 (2018) 129-150.

[6] J. Berlińska, M. Drozdowski, Comparing load-balancing algorithms for MapReduce under Zipfian data skews, Parallel Computing, 72 (2018) 14-28.

[7] D.G. Bonett, Robust confidence interval for a ratio of standard deviations, Applied psychological measurement, 30 (2006) 432-439.

[8] Q. Chen, J. Yao, Z. Xiao, LIBRA: Lightweight Data Skew Mitigation in MapReduce, IEEE Transactions on Parallel and Distributed Systems, 26 (2015) 2520-2533.

[9] J. Dean, S. Ghemawat, MapReduce: simplified data processing on large clusters, Commun. ACM, 51 (2008) 107-113.

[10] S. Del Río, V. López, J.M. Benítez, F. Herrera, On the use of MapReduce for imbalanced big data using Random Forest, Information Sciences, 285 (2014) 112-137.

[11] Y. Fan, W. Wu, Y. Xu, H. Chen, Improving MapReduce performance by balancing skewed loads, China Communications, 11 (2014) 85-108.

[12] E. Friedman, S. Shenker, Synchronous and asynchronous learning by responsive learning automata, Mimeo, (1996).

[13] Y. Gao, Y. Zhang, H. Wang, J. Li, H. Gao, A Distributed Load Balance Algorithm of MapReduce for Data Quality Detection, in: H. Gao, J. Kim, Y. Sakurai (Eds.) Database Systems for Advanced Applications: DASFAA 2016 International Workshops: BDMS, BDQM, MoI, and SeCoP, Dallas, TX, USA, April 16-19, 2016, Proceedings, Springer International Publishing, Cham, 2016, pp. 294-306.

[14] E. Gavagsaz, A. Rezaee, H. Haj Seyyed Javadi, Load balancing in reducers for skewed data in MapReduce systems by using scalable simple random sampling, The Journal of Supercomputing, 74 (2018) 3415-3440.

[15] E. Gavagsaz, A. Rezaee, H. Haj Seyyed Javadi, Load balancing in join algorithms for skewed data in MapReduce systems, The Journal of Supercomputing, (2018).

[16] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, H. Pirahesh, Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, Data Mining and Knowledge Discovery, 1 (1997) 29-53.

[17] B. Gufler, N. Augsten, A. Reiser, A. Kemper, Load balancing in mapreduce based on scalable cardinality estimates, in: 2012 IEEE 28th International Conference on Data Engineering, IEEE, 2012, pp. 522-533.

[18] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, L. Qi, LEEN: Locality/Fairness-Aware Key Partitioning for MapReduce in the Cloud, in: Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on, 2010, pp. 17-24.

[19] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, S. Wu, Handling partitioning skew in MapReduce using LEEN, Peer-to-Peer Networking and Applications, 6 (2013) 409-424.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, A. Liguori, {kvm: the Linux virtual machine monitor}, in: Proceedings of the Linux Symposium, 2007, pp. 225-230.

[21] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, A study of skew in mapreduce applications, Open Cirrus Summit, (2011).

[22] Y. Kwon, M. Balazinska, B. Howe, J. Rolia, SkewTune: mitigating skew in mapreduce applications, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, Scottsdale, Arizona, USA, 2012, pp. 25-36.

[23] J. Li, Y. Liu, J. Pan, P. Zhang, W. Chen, L. Wang, Map-Balance-Reduce: An improved parallel programming model for load balancing of MapReduce, Future Generation Computer Systems, (2017).

[24] J. Lin, The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce, in: 7th Workshop on Large-Scale Distributed Systems for Information Retrieval, 2009.

[25] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, P. Valduriez, FP-Hadoop: Efficient processing of skewed MapReduce jobs, Information Systems, 60 (2016) 69-84.

[26] G. Liu, X. Zhu, J. Wang, D. Guo, W. Bao, H. Guo, SP-Partitioner: A novel partition method to handle intermediate data skew in spark streaming, Future Generation Computer Systems, (2017).

[27] W. Lu, L. Chen, L. Wang, H. Yuan, W. Xing, Y. Yang, NPIY : A novel partitioner for improving mapreduce performance, Journal of Visual Languages & Computing, 46 (2018) 1-11.

[28] B. Memishi, M.S. Pérez, G. Antoniu, Failure detector abstractions for MapReduce-based systems, Information Sciences, 379 (2017) 112-127.

[29] J. Myung, J. Shim, J. Yeon, S.-g. Lee, Handling data skew in join algorithms using MapReduce, Expert Systems with Applications, 51 (2016) 286-299.

[30] K. Najim, A.S. Poznyak, Learning automata: theory and applications, Pergamon Press, Inc., 1994.

[31] K.S. Narendra, M.A.L. Thathachar, Learning automata: an introduction, Prentice-Hall, Inc., 1989.

[32] C.L. Philip Chen, C.-Y. Zhang, Data-intensive applications, challenges, techniques and technologies: A survey on Big Data, Information Sciences, 275 (2014) 314-347.

[33] S.R. Ramakrishnan, G. Swart, A. Urmanov, Balancing reducer skew in MapReduce workloads using progressive sampling, in: Proceedings of the Third ACM Symposium on Cloud Computing, ACM, San Jose, California, 2012, pp. 1-14.

[34] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, D. Reeves, Sailfish: a framework for large scale data processing, in: Proceedings of the Third ACM Symposium on Cloud Computing, ACM, San Jose, California, 2012, pp. 1-14.

[35] K. Slagter, C.-H. Hsu, Y.-C. Chung, G. Yi, SmartJoin: a network-aware multiway join for MapReduce, Cluster Computing, 17 (2014) 629-641.

[36] B. Tang, M. Tang, G. Fedak, H. He, Availability/Network-aware MapReduce over the Internet, Information Sciences, 379 (2017) 94-111.

[37] Z. Tang, X. Zhang, K. Li, K. Li, An intermediate data placement algorithm for load balancing in Spark computing environment, Future Generation Computer Systems, (2016).

[38] M.A.L. Thathachar, B.R. Harita, Learning automata with changing number of actions, IEEE Transactions on Systems, Man, and Cybernetics, 17 (1987) 1095-1100.

[39] M.A.L. Thathachar, P.S. Sastry, Varieties of learning automata: an overview, IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics), 32 (2002) 711-722.

[40] M.A.L. Thathachar, P.S. Sastry, Networks of Learning Automata: Techniques for Online Stochastic Optimization, Springer-Verlag New York, Inc., 2003.

[41] T. White, Hadoop: The Definitive Guide, 4th Edition, O'Reilly Media, Inc., 2015.

[42] Y. Xu, P. Zou, W. Qu, Z. Li, K. Li, X. Cui, Sampling-Based Partitioning in MapReduce for Skewed Data, in: 2012 Seventh ChinaGrid Annual Conference, 2012, pp. 1-8.

[43] Y. Xu, W. Qu, Z. Li, Z. Liu, C. Ji, Y. Li, H. Li, Balancing reducer workload for skewed data using sampling-based partitioning, Computers & Electrical Engineering, 40 (2014) 675-687.

[44] X. Zhang, Y. Wu, C. Zhao, MrHeter: improving MapReduce performance in heterogeneous environments, Cluster Computing, 19 (2016) 1691-1701.

[45] X. Zhao, J. Zhang, X. Qin, k NN-DP: Handling Data Skewness in kNN Joins Using MapReduce, IEEE Transactions on Parallel and Distributed Systems, 29 (2018) 600-613.