Peer-to-Peer Netw. Appl.
DOI 10.1007/s12083-013-0213-7

Handling partitioning skew in MapReduce using *LEEN*

Shadi Ibrahim · Hai Jin · Lu Lu · Bingsheng He · Gabriel Antoniu · Song Wu

Received: 1 November 2012 / Accepted: 24 April 2013

Abstract MapReduce is emerging as a prominent tool for big data processing. Data locality is a key feature in MapReduce that is extensively leveraged in data-intensive cloud systems: it avoids network saturation when processing large amounts of data by co-allocating computation and data storage, particularly for the map phase. However, our studies with Hadoop, a widely used MapReduce implementation, demonstrate that the presence of *partitioning skew* (Partitioning skew refers to the case when a variation in either the intermediate keys' frequencies or their distributions or both among different data nodes) huge amount of data transfer during the shuffle phase and leads to significant unfairness on the reduce input among different data nodes. As a result, the applications severe performance degradation due to the long data transfer during the shuffle phase along with the computation skew, particularly in reduce phase. In this paper, we develop a novel algorithm named *LEEN* for locality-aware and fairness-aware key partitioning in MapReduce. *LEEN* embraces an asynchronous map and reduce scheme. All buffered intermediate keys are partitioned according to their frequencies and the fairness of the expected data distribution after the shuffle phase. We have integrated *LEEN* into Hadoop. Our experiments demonstrate that *LEEN* can efficiently achieve higher locality and reduce the amount of shuffled data. More importantly, *LEEN* guarantees fair distribution of the reduce inputs. As a result, *LEEN* achieves a performance improvement of up to 45 % on different workloads.

Keywords MapReduce · Hadoop · Cloud computing · Skew partitioning · Intermediate data

1 Introduction

MapReduce [1], due to its remarkable features in simplicity, fault tolerance, and scalability, is by far the most successful realization of data intensive cloud computing platforms [2]. It is often advocated as an easy-to-use, efficient and reliable replacement for the traditional programming model of moving the data to the cloud [3]. Many implementations have been developed in different programming languages for various purposes [4–6]. The popular open source implementation of MapReduce, Hadoop [7], was developed primarily by Yahoo, where it processes hundreds of terabytes of data on tens of thousands of nodes [8], and is now used by other companies, including Facebook, Amazon, Last.fm, and the New York Times [9].

The MapReduce system runs on top of the Google File System (GFS) [10], within which data is loaded, partitioned into chunks, and each chunk replicated across multiple machines. Data processing is co-located with data storage: when a file needs to be processed, the job scheduler consults a storage metadata service to get the host node for each

S. Ibrahim (✉) · G. Antoniu
INRIA Rennes-Bretagne Atlantique, Rennes, France
e-mail: shadi.ibrahim@inria.fr

G. Antoniu
e-mail: gabriel.antoniu@inria.fr

H. Jin · L. Lu · S. Wu
Cluster and Grid Computing Lab, Services Computing
Technology and System Lab, Huazhong University of Science
and Technology, Wuhan, China

H. Jin
e-mail: hjin@mail.hust.edu.cn

B. He
School of Computer Engineering, Nanyang Technological
University, Singapore, Singapore
e-mail: bshe@ntu.edu.sg

Published online: 25 May 2013

chunk, and then schedules a "map" process on that node, so that data locality is exploited efficiently. The map function processes a data chunk into key/value pairs, on which a hash partitioning function is performed, on the appearance of each intermediate key produced by any running map within the MapReduce system:

hash (hash code (Intermediate-Keys) module ReduceID)

The hashing results are stored in memory buffers, before spilling the intermediate data (index file and data file) to the local disk [11]. In the reduce stage, a reducer takes a partition as input, and performs the reduce function on the partition (such as aggregation). Naturally, how the hash partitions are stored among machines affects the network traffic, and the balance of the hash partition size is an important indicator for load balancing among reducers.

In this work, we address the problem of how to efficiently partition the intermediate keys to decrease the amount of shuffled data, and guarantee fair distribution of the reducers' inputs, resulting in improving the overall performance. While, the current Hadoop's hash partitioning works well when the keys are equally appeared and uniformly stored in the data nodes, with the presence of partitioning skew, the blindly hash-partitioning is inadequate and can lead to:

1. Network congestion caused by the huge amount of shuffled data, (for example, in wordcount application, the intermediate data are 1.7 times greater in size than the maps input, thus tackling the network congestion by locality-aware map executions in MapReduce systems is not enough);
2. unfairness of reducers' inputs; and finally
3. severe performance degradation [12] (i.e. the variance of reducers' inputs, in turn, causes a variation in the execution time of reduce tasks, resulting in longer response time of the whole job, as the job's response time is dominated by the slowest reduce instance).

Recent research has reported on the existence of partitioning skew in many MapReduce applications [12–14], but none of the current MapReduce implementations have overlooked the data skew issue [15]. Accordingly, in the presence of partitioning skew, the existing shuffle strategy encounters the problems of long intermediate data shuffle time and noticeable network overhead. To overcome the network congestion during the shuffle phase, we propose to expose the locality-aware concept to the reduce task; However, locality-aware reduce execution might not be able to outperform the native MapReduce due to the penalties of unfairness of data distribution after the shuffle phase, resulting in reduce computation skew. To remedy this deficiency, we have developed an innovative approach to significantly reduce data transfer while balancing the data distribution among data nodes.

Recognizing that the network congestion and unfairness distribution of reducers' inputs, we seek to reduce the transferred data during the shuffle phase, as well as achieving a more balanced system. We develop an algorithm, locality-aware and fairness-aware key partitioning (*LEEN*), to save the network bandwidth dissipation during the shuffle phase of the MapReduce job along with balancing the reducers' inputs. *LEEN* is conducive to improve the data locality of the MapReduce execution efficiency by the virtue of the asynchronous map and reduce scheme, thereby having more control on the keys distribution in each data node. *LEEN* keeps track of the frequencies of buffered keys hosted by each data node. In doing so, *LEEN* efficiently moves buffered intermediate keys to the destination considering the location of the high frequencies along with fair distribution of reducers' inputs. To quantify the locality, data distribution and performance of *LEEN*, we conduct a comprehensive performance evaluation study using *LEEN* in Hadoop 0.21.0. Our experimental results demonstrate that *LEEN* interestingly can efficiently achieve higher locality, and balance data distribution after the shuffle phase. In addition, *LEEN* performs well across several metrics, with different partitioning skew degrees, which contribute to the performance improvement up to 45 %.

LEEN is generally applicable to other applications with data partitioning and this will result in guaranteed resource load balancing with a small overhead due to the asynchronous design. The main focus of this paper and the primary usage for *LEEN* is on MapReduce applications where partitions skew exists (e.g., many scientific applications [12–14, 16] and graph applications [17]).

We summarize the contributions of our paper as follows:

- An in-depth study on the source of partitioning skew in MapReduce and its impacts on application performance.
- A natural extension of the data-aware execution by the native MapReduce model to the reduce task.
- A novel algorithm to explore the data locality and fairness distribution of intermediate data during and after the shuffle phase, to reduce network congestion and achieve acceptable data distribution fairness.
- Practical insight and solution to the problems of network congestion and reduce computation skew, caused by the partitioning skew, in emerging Cloud.

The rest of this paper is organized as follows. Section 2 briefly introduces MapReduce and Hadoop, and illustrates the recent partitioning strategy used in Hadoop. The partitioning skew issue is explored and empirically analyzed in Section 3. The design and implementation of the *LEEN* approach is discussed in Section 4. Section 5 details the performance evaluation. Section 6 discusses the related works. Finally, we conclude the paper and propose our future work in Section 7.

2 Background

In this section, we briefly introduce the MapReduce model and its widely used implementation, Hadoop. Then we briefly zoom on the workflow of job execution in Hadoop introducing side by side the map, reduce and partition functions.

2.1 MapReduce model

The MapReduce [1] abstraction is inspired by the Map and Reduce functions, which are commonly used in functional languages such as Lisp. Users express the computation using two functions, map and reduce, which can be carried out on subsets of the data in a highly parallel manner. The runtime system is responsible for parallelizing and fault handling.

The steps of the process are as follows:

- The input is read (typically from a distributed file system) and broken up into key/value pairs. The key identifies the subset of data, and the value will have computation performed on it. The map function maps this data into sets of key/value pairs that can be distributed to different processors.
- The pairs are partitioned into groups for processing, and are sorted according to their key as they arrive for reduction. The key/value pairs are reduced, once for each unique key in the sorted list, to produce a combined result.

2.2 Hadoop

Hadoop [7] is a java open source implementation of MapReduce sponsored by Yahoo! The Hadoop project is a collection of various subprojects for reliable, scalable distributed computing. The two fundamental subprojects are the Hadoop MapReduce framework and the HDFS. HDFS is a distributed file system that provides high throughput access to application data [7]. It is inspired by the GFS. HDFS has master/slave architecture. The master server, called NameNode, splits files into blocks and distributes them across the cluster with replications for fault tolerance. It holds all metadata information about stored files. The HDFS slaves, the actual store of the data blocks called DataNodes, serve read/write requests from clients and propagate replication tasks as directed by the NameNode.

The Hadoop MapReduce is a software framework for distributed processing of large data sets on compute clusters [7]. It runs on the top of the HDFS. Thus data processing is collocated with data storage. It also has master/slave architecture. The master, called Job Tracker (JT), is responsible of: (a) Querying the NameNode for the block locations, (b) considering the information retrieved by the NameNode, JT schedule the tasks on the slaves, called Task Trackers (TT), and (c) monitoring the success and failures of the tasks.

2.3 Zoom on job execution in Hadoop

The MapReduce program is divided into two phases, map and reduce. For the map side, it starts by reading the records in the Map process, then the map function processes a data chunk into key/value pairs, on which the hash partitioning function is performed as shown in Fig. 1. This intermediate result, refereed as record, is stored with its associate partition in the buffer memory (100 MB for each map by default). If the buffered data reaches the buffer threshold (80 % of the total size), the intermediate data will be sorted according to the partition number and then by key and spilled to the local disk as an index file and a data file. All files will be then merged as one final indexed file—by indexed we mean indexed according to the partition number that represents the target reduce. The reduce case is starting as soon as the intermediate indexed files are fetched to the local disk; the files from multiple local map outputs will be written at the same time (by default five pipes will be available for the different nodes). The files will be buffered in the memory in a "shuffle buffer"; when the shuffle buffer reaches a threshold the files will be redirected to the local disk, then the different files will be merged according to the user specific application, and merged files from the shuffle buffer will be tailed in the local disk. Finally the merged data will be passed to the reduce function and then the output will be written to the HDFS or elsewhere according to the user specific application.

3 Partitioning skew in MapReduce

The outputs of map tasks are distributed among reduce tasks via hash partitioning. The default hash-partitioning, however, is designed to guarantee evenly distribution of keys amongst the different data nodes, that is, if we have n data nodes and k different keys then the number of keys which will be partitioned to each data node is $\frac{k}{n}$, regardless of the frequencies of each distinct key (usually the number of records are associated with one key). The default hash-partitioning therefore is only adequate when the number of records associated with each key are relatively equal and the key's records are uniformly distrusted amongst data nodes.

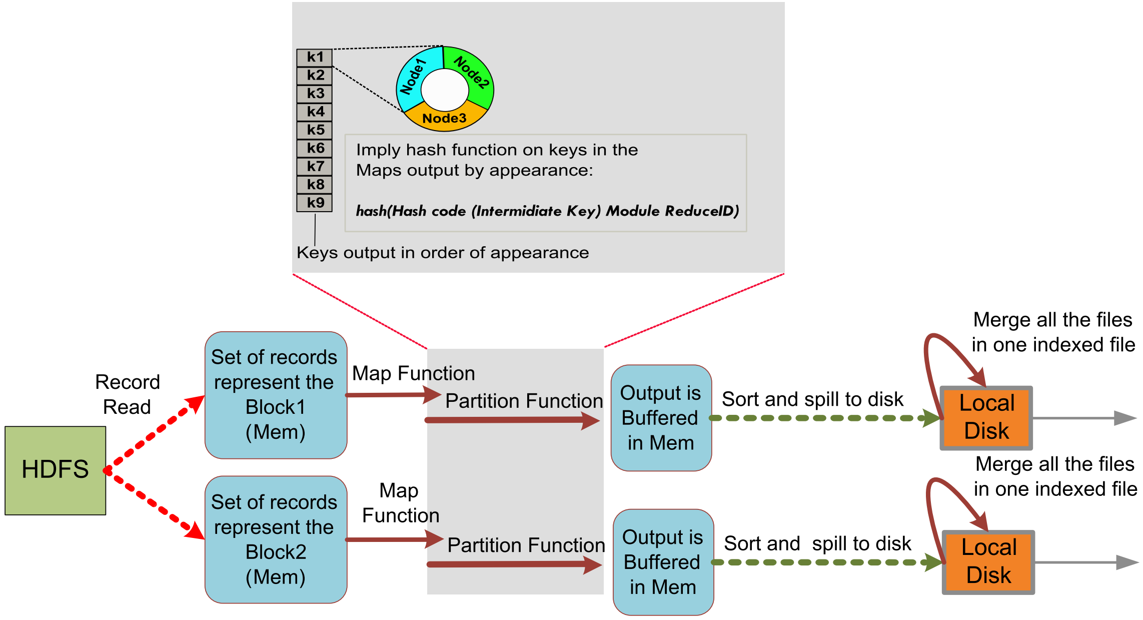
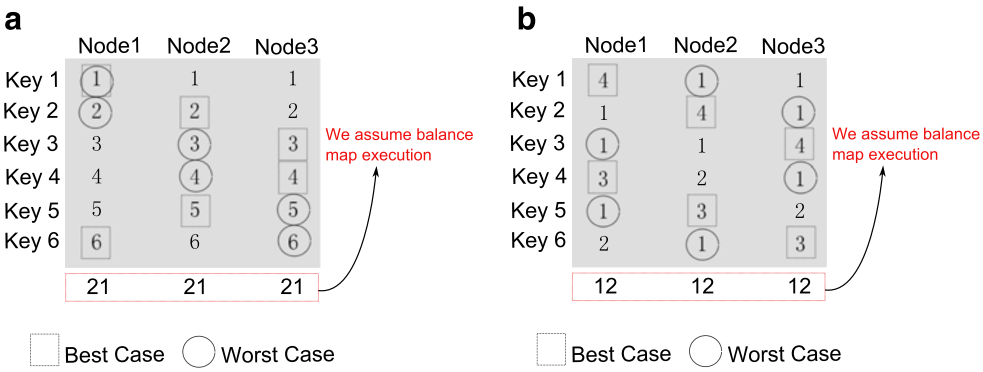

k1
k2
k3
k4
k5
k6
k7
k8
k9
Node1
Node2
Node3
Imply hash function on keys in the Maps output by appearance:
hash(Hash code (Intermidiate Key) Module ReduceID)
Keys output in order of appearance
HDFS
Record Read
Set of records represent the Block1 (Mem)
Set of records represent the Block2 (Mem)
Map Function
Partition Function
Map Function
Partition Function
Output is Buffered in Mem
Output is Buffered in Mem
Sort and spill to disk
Sort and spill to disk
Merge all the files in one indexed file
Merge all the files in one indexed file
Local Disk
Local Disk

Fig. 1 The workflow of the two phases in MapReduce job: the map phase and reduce phase

However, in the presence of partitioning skew the hash-partitioning assumption will break and therefore reduce-skew and network congestion can arise in practice [12–14, 18]. As we earlier stated the partition skew phenomena refereed to the case when the keys' frequencies vary and/or the key's records among data node are not uniformly distributed. Consider the two examples which represent each factor separately:

- Keys' Frequencies Variation: Although the partitioning function perfectly distributes keys across reducers, some reducers may still be assigned more data simply because the key groups they are assigned to contain significantly more values. Figure 2a presents the first example considering three data nodes and six keys. We vary keys frequencies to 3, 6, 9, 12, 15, *and*18 records per key, accordingly using the blindly hash-partitioning which is based on the sequence of the keys appearance during the map phase, the distribution of reducers' inputs will vary between the best partitioning: 21 records for each reducer, and the worst case partitioning: the input of the reducers in `node1`, `node2`, and `node3` will be 9, 21 and 33 records respectively. Despite that in both cases the number of keys assigned to each data node is the same, two keys per node in our example.

Accordingly reduce-skew will occur, in our example, `node3` will finish its reduce nearly four times slower than node1; consequently, heavy reduce execution on some nodes. Thus performance experiences degradation


a
b
We assume balance map execution
We assume balance map execution

a

	Node1	Node2	Node3
Key 1	○1	1	1
Key 2	○2	□2	2
Key 3	3	○3	□3
Key 4	4	○4	□4
Key 5	5	□5	○5
Key 6	□6	6	○6
	21	21	21

□ Best Case ○ Worst Case

b

	Node1	Node2	Node3
Key 1	□4	○1	1
Key 2	1	□4	○1
Key 3	○1	1	□4
Key 4	□3	2	○1
Key 5	○1	□3	2
Key 6	2	○1	□3
	12	12	12

□ Best Case ○ Worst Case

Fig. 2 Motivational example: demonstrates the worst and best partitioning scenarios when applying the current blindly key partitioning in MapReduce in the presence of Partitioning skew. The keys are ordered by their appearance while each value represents the frequency of the key in the data node. **a** Keys's frequencies variation. **b** Inconsistency in key's distribution

(i.e. waiting the last subtask to be finished), and less resource utilization (i.e. node1 will be idle while node3 is overloaded). Schatz [18] and Kwon [12] have demonstrated the existence of this phoneme in some biological applications, for example, Kwon [12] has demonstrated that because of the keys' frequencies variation, in CloudBurst [18] application, some reducers will finish their task four times longer than other reduces.

- Inconsistency in Key's Distribution: As a second example, even when the keys have the same frequencies and therefore the partitioning function perfectly distributes keys across reducers—all reducers inputs are relatively equal–. But, however, the blind hash-partitioning may lead to high network congestion, especially when the key's recodes are not uniformly distributed among data nodes. Figure 2b presents the second example considering three data nodes and six keys. All keys have the same frequents, 6 records per key but the key's distribution is inconsistent among the nodes. Applying the blindly hash-partitioning will result with evenly reducers's inputs, but the data transfer, in contrast with the total map output during the shuffle phase will vary from 41.6 %,[1] in the best case, to 83.3 % in the worst case. Accordingly network congestion during the shuffle phase is strongly depending on the hash-partitioning.

However, in the case of partitioning skew, when both factors, keys' frequencies variation and inconsistency in key's distribution, will occur the blind hash-partitioning may result will both skew-reduce and network congestion as demonstrated in Section 5.

3.1 Partitioning skew in MapReduce applications

MapReduce has been applied widely in various fields including data- and compute-intensive applications, machine learning, and multi-core programming. In this subsection we intend to classify the MapReduce application in term of skewed intermediate data.

A typical MapReduce application includes four main functions: map, reduce, combiner and shuffle functions. Accordingly we could classify MapRduce applications in respect to the main applied function in these applications into: map-oriented, combiner-oriented, map/reduce-oriented , shuffle-oriented as shown in Table 1.

- **Map-oriented.** The map function is the main function in the application, while the reduce function is only an identity function. An example of this type of applications is the Distributed Grep application.[2]
- **Combiner-oriented.** The combiner function is applied in such applications. The combiner performs as a map-based pre-reducer which significantly reduces the network congestion as in wordcount[3] applications and Count of URL Access Frequency.[4]
- **Map/Reduce-oriented.** These applications are typical map and reduce jobs where no combine can be applied. Also in this type of applications, all the keys is associated with only one unique value as in distributed Sort.[5]
- **Shuffle-oriented.** In these applications both map and reduce functions are applied. However, they differ from the previous application in that multi record are associated with the same key and they differ from the second type in that no combiner could be used. Here when the map output is shuffled to the reducer, this may cause a network bottleneck. There is a wide range of applications in this category as graph processing, machine learning and scientific application [12, 16, 18–20]. It is important to note that many optimizations could be applied in this category.

3.2 Empirical study on partitioning skew in Hadoop

In this section we empirically demonstrate the impacts of the partition skew on MapReduce applications. For simplicity, we mimic the first type of partitioning skew, frequencies variation, which was in practise in some real applications. We use wordcount benchmark but after disabling the combiner function.

3.2.1 Experimental environment

Our experimental hardware consists of a cluster with four nodes. Each node is equipped with four quad-core 2.33 GHz Xeon processors, 24 GB of memory and 1 TB of disk, runs RHEL5 with kernel 2.6.22, and is connected with 1 GB Ethernet. In order to extend our testbed, we use a virtualized environment, using Xen [22]. In the virtualized environment, one virtual machine (VM) was deployed on one physical machine (PM) to act as master node (Namenode). We also deployed two VMs on each of the three left PM, reaching a cluster size of 6 data nodes. Each virtual machine is configured with 1 CPU and 1 GB memory.

All results described in this paper are obtained using Hadoop-0.21.0. In order to show the case of partitioning

[1] This value represents the ratio= $\frac{transferreddata\ during\ shuffle}{map\ phase\ output}$

[2] http://wiki.apache.org/hadoop/Grep

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

[4] http://code.google.com/intl/fr/edu/parallel/mapreduce-tutorial.html

[5] http://wiki.apache.org/hadoop/Sort

Table 1 MapReduce applications' classification

Map only	MapReduce	MapReduce without combiner	
	with combiner	Single record	Multi record
Distributed grep	Wordcount	Distributed sort	Wordcount without combiner,
	Count of URL access frequency		Graph processing [21],
			Machine learning [20], Scientific application [16, 18, 19]

skew, we perform the wordcount applications without combiner function. Moreover, we have used up to 100 different keys reaching an input data size of 6 GB: representing different words with the same length (to avoid variation in values size), with different frequencies as shown in Fig. 3 (we vary the keys frequencies between 60 to 79860000 records), and uniform key distribution between nodes: if a key frequency is 60, then each data node is hosting 10 records of this key.

3.2.2 Major results

As we mentioned earlier, the current partition function blindly partitions the keys to the available reducers: it ignores the keys' frequencies variation and their distribution. This in turn will lead to skewed reducers inputs and also reduce computation skew.

As shown in Fig. 4a, although the keys are uniformly distributed between the nodes (the data locality of shuffled keys is fixed to $\frac{1}{n}$, where n is the number of nodes "16 %"), we observe a huge amount of data transfer during the shuffle phase (almost 14.7 GB) which is by far greater than the input data (6 GB). This supports our motivation on shuffled data being an important source of network saturation in MapReduce applications. Moreover, we observe an imbalanced network traffic among the different data nodes: some nodes will suffer heavy network traffic while low traffic in other nodes.

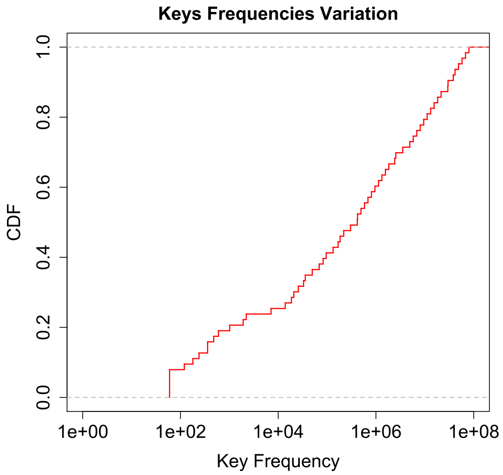


Fig. 3 Experiment setup: CDF of the Keys' frequencies. The key frequencies vary from 60 to 79860000 records per key

Moreover, the data distribution of reducers inputs is totally imbalanced: it ranges from 340 MB to 3401 MB as shown in Fig. 4b, which in turn will result in a reduce computation skew as shown in Fig. 5. As the minimum size of reducer input (`node1`) is almost 10 % compared to the maximum one (`node6`), this will result with misuse of the system resources: for example one `node1` will finish processing the reduce function nearly nine times faster than `node6` (`node1` finishes the reduce function in 33 s while `node6` finishes in 231 s). Accordingly some nodes will be heavily overloaded while other nodes are idle.

As a result the application experiences performance degradation: waiting for the last task to be completed.

4 LEEN: Locality-awarE and fairness-awarE key partitioNing

To address the partitioning skew problem and limit its adversary's impacts in MapReduce: Network saturation and imbalanced reduce execution, in this section we propose a new key partitioning approach that exposes data locality to the reduce phase while maintaining fair distribution among the reducers' inputs. We first discuss the asynchronous map and reduce scheme (Section 4.1), later we discuss in details the *LEEN* algorithm (Section 4.2) and finally we describe the implementation of *LEEN* in Hadoop (Section 4.3).

4.1 Asynchronous map and reduce

In Hadoop several maps and reduces are concurrently running on each data node (two of each by default) to overlap computation and data transfer. While in *LEEN*, in order to keep a track on all the intermediate keys' frequencies and

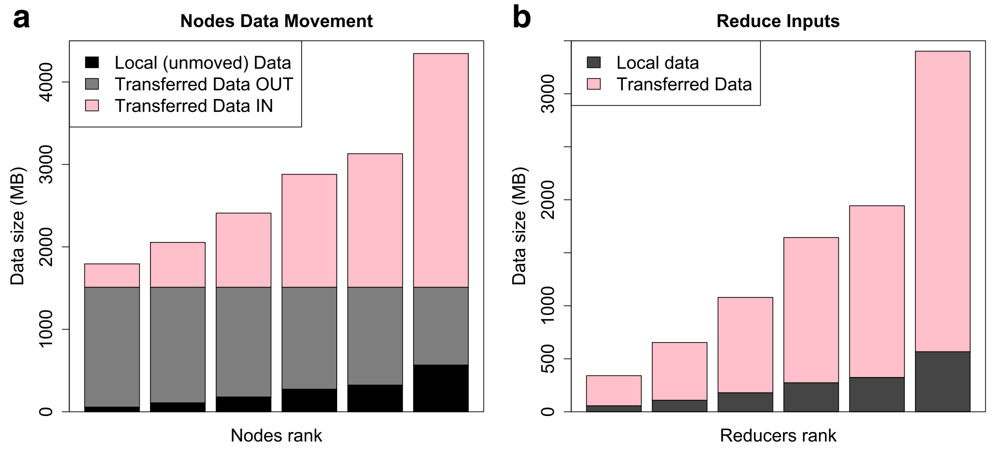


Fig. 4 The size of data transferred from and into the data nodes during the copy phase and the data distribution of reducers inputs: when performing *wordcount* application on 6 GB of data after disabling the combiners. **a** Data movement during shuffle phase: although the number of keys per reducer task is the same, the data transferred in and out vary in accordance to the number of records per key. **b** The data distribution of reducers inputs: even though all reduce tasks receive the same number of keys, the size of reducers inputs varies from 340MB to 3401MB

key's distributions, we propose to use asynchronous map and reduce schemes, which is a trade-off between improving the data locality along with fair distribution and concurrent MapReduce, (concurrent execution of map phase and reduce phase). Although, this trade-off seemed to bring a little overhead due to the unutilized network during the map phase, but it can fasten the map execution because the complete I/O disk resources will be reserved to the map tasks. For example, the average execution time of map tasks when using the asynchronous MapReduce was 26 s while it is 32 s in the native Hadoop. Moreover, the speedup of map executions can be increased by reserving more memory for buffered maps within the data node. This will be beneficial, especially in the Cloud, when the executing unit is a VM with a small memory size (e.g. In Amazon EC2 [23], the small instance has 1 GB of virtual memory). In our scheme, when the map function is applied on input record, similar to the current MapReduce, a partition function will be applied on the intermediate key in the buffer memory by their appearance in the maps output, but the partition number represents a unique ID which is the KeyID:

hash (hash code (Intermediate-Keys) module KeyID)

Thus, the intermediate data will be written to the disk as an index file and data file, each file represents one key, accompanied by a metadata file, DataNode-Keys Frequency Table, which include the number of the records in each file, represent the key frequency. Finally, when all the Maps are done all the metadata files will be aggregated by the Job Tracker then the keys will be partitioned to the different data nodes according to the *LEEN* algorithm.

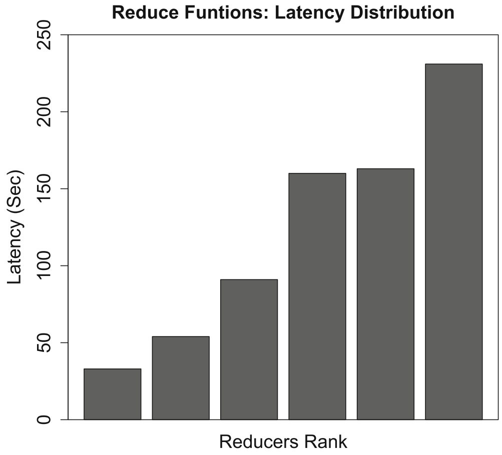


Fig. 5 Reducers function latency: there is a factor of nine difference in latency between the fastest and the slowest reduce functions which is due to the reducers inputs skew

4.2 LEEN algorithm

In this section, we present our *LEEN* algorithm for locality-aware and fairness-aware key partitioning in MapReduce. In order to effectively partition a given data set of K keys, distributed on N data nodes, obviously, we need to find the best solution in a space of K^N of possible solutions, which is too large to explore. Therefore, in *LEEN*, we use a heuristic method to find the best node for partitioning a specific key, then we move on to the second key. Therefore, it is important that keys are sorted. *LEEN* is intending to provide a solution which provides a close to optimal tradeoff between data locality and reducers' input fairness, that is, to provide a solution where the locality of the keys partitioning achieve maximum value while keeping in mind the best fairness of reducers' input (smallest variation). Thus the solution achieves minimum value of the $\frac{Fairness}{Locality}$. Locality is the sum of keys frequencies in the nodes—which are partitioned to—to the total keys frequencies.

$$Locality_{LEEN} = \frac{\sum_{i=1}^{K} FK_i^j}{\sum_{i=1}^{K} FK_i} \tag{1}$$

Where FK_i^j indicate the frequency of key k_i in the data node n^j, if k_i partitioned to n^j, and FK_i represents the total frequency of key k_i, which is the sum of the frequencies of k_i in all the data nodes: $FK_i = \sum_{j=1}^{nodes} FK_i^j$. And the locality in our system will be bounded by:

$$\frac{\sum_{i=1}^{K} min_{1\leq j\leq n} FK_i^j}{\sum_{i=1}^{K} FK_i} < Locality_{LEEN} < \frac{\sum_{i=1}^{K} max_{1\leq j\leq n} FK_i^j}{\sum_{i=1}^{M} FK_i} \tag{2}$$

Fairness is the variation of the reducers' inputs. In MapReduce systems, the response time is dominated by the slowest sub-task, in our case the slowest reduce task, therefore, in terms of performance score the fairness of *LEEN* can be presented by the extra data of the maximum reducers' inputs to the average, called overload data, refereed as $D_{overload}$:

$$\begin{aligned} D_{overload} &= max(Reducers\ input) - Mean \\ &= max(HostedDataN_K^j) - Mean \end{aligned} \tag{3}$$

Where $HostedDataN_K^j$ is the data hosted in node n^j after partitioning all the K keys.

$$HostedDataN_i^j = \begin{cases} SumKN^j & \text{, the initial value} \\ HostedDataN_{i-1}^j + (FK_i - FK_i^j) & \text{, } k_i \text{ is partitioned to } n^j \\ HostedDataN_{i-1}^j - FK_i^j & \text{, } k_i \text{ is not } partitioned \text{ to } n^j \end{cases} \tag{4}$$

Where $SumKN^j$ represents the sum of the all keys frequencies within that data node n^j: $SumKN^j = \sum_{i=1}^{Keys} FK_i^j$. When processing keys in *LEEN*, it is important that the keys are sorted. Thus we sort the keys according to their $\frac{Fairness}{Locality}$ values. As keys with small value will have less impact on the global $\frac{Fairness}{Locality}$, therefore, we sort the keys in descending order according to their Fairness-locality value, refereed as FLK.

$$FLK_i = \frac{Fairness\ in\ distribution\ of\ K_i\ amongst\ data\ nodes}{Best\ Locality} \tag{5}$$

The fairness of the key distribution is presented by using the standard deviation of this key and refereed as $DevK_i$.

$$DevK_i = \sqrt{\frac{\sum_{j=1}^{n}(FK_i^j - Mean)^2}{N}} \tag{6}$$

Where FK_i^j indicate the frequency of key k_i in the data node n^j, and Mean represents the mean of FK_i^j values. Obviously, the best locality indicate partitioning k_i to the data node n^j which has the maximum frequencies. FLK_i can be formulated as:

$$FLK_i = \frac{DevK_i}{max_{1\leq j\leq n} FK_i^j} \tag{7}$$

Initially, the hosted data on each node is set to their initial values, with the assumption of equal maps outputs, the initial value of hosted data on each node are equal and can be presented as $\frac{total\ data}{the\ number\ of\ datanodes}$. For a specific key in order to achieve the best locality, we select the node with maximum frequency, therefore, we sort the nodes in descending order according to their $FK_i^j(s)$. Then we compare the current node with the next node (second maximum frequency). If the Fairness-Score, which is the variation of the expected hosted data among all the data nodes if this key will be partitioned to this node, of the second node is better than the current one, it is accepted. *LEEN* recursively tries the next lower node. The node is determined when the new node fairness-score is worse than the current one. After selecting the node, it moves on to the next key and calculates the new values of hosted data in the different data nodes $HostedDataN_i^j$. The fairness-Score is defined as:

$$Fairness - Score_i^j = \sqrt{\frac{\sum_{j=1}^{n}(HosteddataN_i^j - Mean)^2}{N}} \tag{8}$$

It is very important that our heuristic method has running time at most $K \times N$. In general, the overhead of the *LEEN* algorithm is negligible at small/medium scale (for example, in our experiments, the overhead of *LEEN* when partitioning 1,000 keys to 30 VMs was beyond 1 s). However, to deal with large scale problems we introduce the concept of Virtual Key (VK), this will be discussed further in Section 4.3.

The complete algorithm is represented in Algorithm 1.

Algorithm 1 LEEN Algorithm

```
Input: K: set of Keys and N: the number of data nodes;
Description: perform partition function on a set of keys, with different frequencies to different data nodes. The keys are sorted in descending order according to their $FLK_i$ values.
Output: partition $(k_i, n^j)$;
for $k_i \in K$ do
    process the nodes according to their $FK_i^j$.
    $j \leftarrow 0$;
    while $Fairness-ScoreN_i^j > Fairness-ScoreN_i^{j+1}$ do
        $j \leftarrow j+1$;
    end
    partition $(k_i, n^j)$
    for $n^j \in N$ do
        Calculate $HostedDataN_i^j$
    end
end
```

4.3 *LEEN*-Hadoop implementation

The *LEEN* core scheduling algorithm is implemented in the *cn.edu.hust.grid.leen* package. Adopting Hadoop to work with *LEEN* requires modifications to the Hadoop source code. The changes relate to expose the key frequency statistics and manage the shuffle I/O as shown in Fig. 6.

In particular, in the Task Tracker, we change the job partition number from the reduce task number to the virtual key number;[6] therefore making the default hash partition function grouping the records that have the same virtual key into the same partition. Then the `collect()` method records the numbers of key-value pairs for each virtual key. After the Job Tracker marks a map task successfully completed, the Task Tracker will send the key frequency statistic to the JobTracker associated with the task completion report. In addition, we modified the Job Tracker to adopt with *LEEN* scheduling behaviors. In particular, to collect the virtual key frequencies of each map task from the task completion report; consequently, when the map phase finishes, all the necessary information is aggregated to form a key distribution matrix as the input of *LEEN*-Algorithm class, and then the `doAlgorithm()` method is invoked to generate the final partition list. This list will be wrapped along with the current Task Tracker list of the Hadoop cluster and sent out to all Task Trackers later.

Moreover, the *LEEN*-Hadoop implementation has two important components for shuffle I/O management:

1. *LEEN Fetcher* manages the shuffle communication of the reducers. The original shuffle class uses several fetcher threads controlled by `ShuffleScheduler` to fetch map outputs from Task Trackers. We modified it to launch one `LeenFetcher` thread per Task Tracker to register itself to and receive intermediate data from the associated Task Tracker. Considering that the reduce task does not need to share system memory space with map tasks, the most merging work is performed inside the shuffle memory buffer.
2. *LEEN* Shuffle Server that replaces the original http-based `MapOutputServlet`. After all the `LeenFetcher` threads register their communication channels to the `LeenShuffleServer`, it will start the shuffle process. It aggregates random disk I/O operations into sequential I/O, thus shuffle manager of Task Tracker sequentially reads map output files and pushes each VK partition to the associated reduce task according to the partition list one by one

5 Performance evaluation

5.1 Experiments setup

LEEN can be applied to Hadoop at different versions. *LEEN* is currently built in Hadoop-0.18.0 (as presented in our previous work [24]) and Hadoop-0.21.0. Our experimental hardware consists of a cluster with seven nodes. Each node is equipped with four quad-core 2.33 GHz Xeon processors, 24 GB of memory and 1TB of disk, runs RHEL5 with kernel 2.6.22, and is connected with 1 GB Ethernet. We evaluate *LEEN* performance in two virtual clusters: on 6VM cluster—similar to the one described in Section 3.2—and on 30VM virtual cluster: one virtual machine (VM) was deployed on one physical machine (PM) to act as the master node (Namenode). We also deploy five VMs on each of the six left PMs, reaching a cluster size of 30 data nodes. All virtual machines are configured with 1 CPU and 1 GB memory.

We conduct our experiments with native Hadoop-0.21.0 and then with *LEEN*. In our experiments using the keys' frequencies variation and the key's distribution are very important parameters in the motivation of the *LEEN* design. While, the keys' frequencies variation will obviously cause

[6] In order to make our system scalable in term of keys number and cluster scale, that to minimize the overhead brought by *LEEN* algorithm, we use the concept Virtual Key (VK) which may in turn be composed of multiple keys. VK is a configurable parameter which can be set by the system administrator.

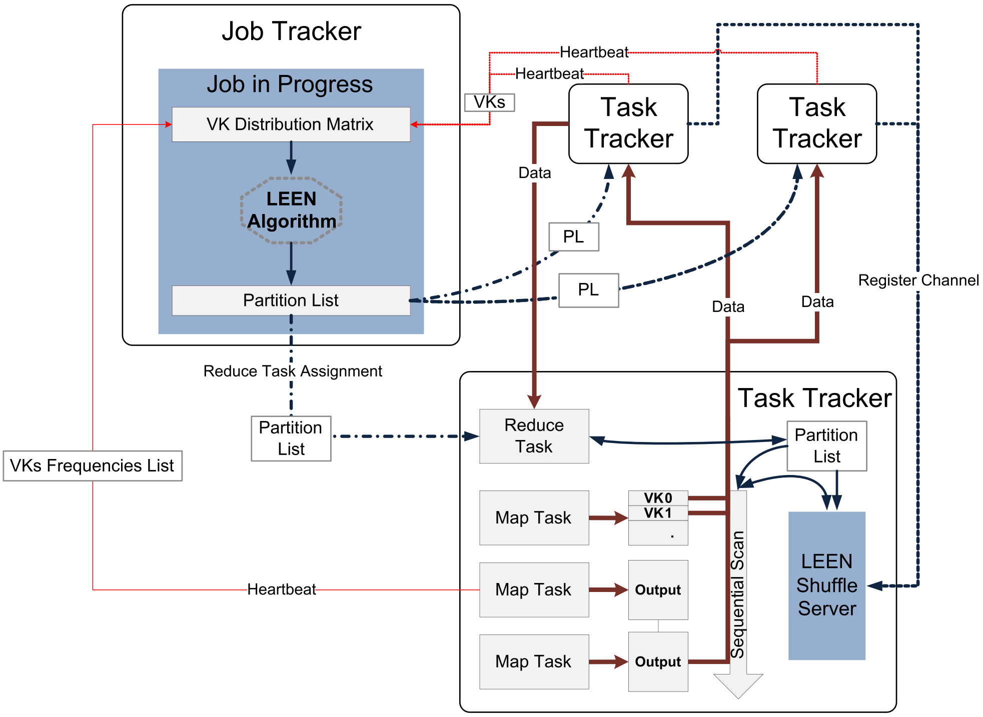


Fig. 6 *LEEN* Architecture

variation of the data distribution of reducers' inputs, the variation in the key's distribution will affect the amount of data transferred during the shuffle phase. To control the keys' frequencies variation and the variation of each key distribution, we modify the existing textwriter code in Hadoop for generating the input data into the HDFS (the number of generated keys varies from 100 to 1,000 keys), and we get three different test sets shown in Table 2. We use primarily the wordcount workload without combiner function as a testing workload.

5.2 Data locality in *LEEN*

We first compare the data locality of reducers' inputs in both native Hadoop and *LEEN*. As shown in Fig. 7, for the first test set (6VM 1), both *LEEN* and native Hadoop achieve the maximum possible locality (16 %). This can be explained due to the uniform distribution of each key among the data nodes (Key distribution variation= 0 %). Here the data locality is depending on the number of data nodes ($Locality = \frac{1}{Number of datanodes}$). For the other two test sets, *LEEN* achieves a higher locality than native Hadoop: the data localities are 55 % and 12.5 % in *LEEN* while they are 11.75 % and 2.5 % in Native Hadoop. While the data locality varies in Hadoop in accordance to the sequence of the key's processing (different run of the same workload may result with different data locality), the data locality

Table 2 Test sets used in the experiments

	6VMs 1	6VMs 2	30VMs
Nodes number	6VMs	6VMS	30VMs
Data size	6 GB	6 GB	7.5 GB
Keys frequencies variation	207 %	44 %	116 %
Key distribution variation (average)	0 %	206 %	130 %
Locality range	16 %	1-69 %	1-16 %

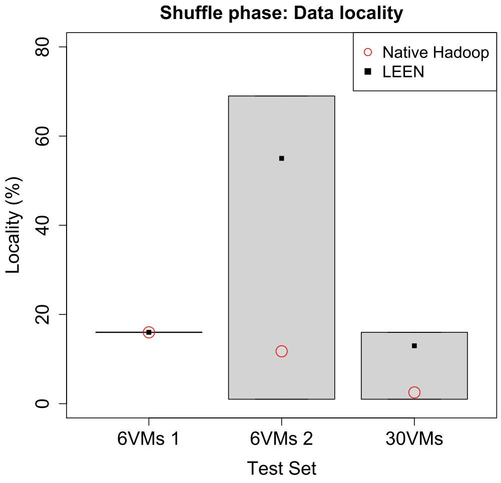


Fig. 7 Data locality in *LEEN* against native Hadoop with different experiments setup: the light gray rectangles represent the locality boundaries that could be achieved in each test (It is calculated using the *Locality_LEEN* boundaries defined in Section 4.2)

in *LEEN* is the same for the same workload it is proportional to the key's variation and varies in accordance to the keys' frequencies variation (*LEEN* is designed to achieve close to optimal tradeoff between data locality and balanced distribution of reducers' inputs).

As a result of the higher data locality in *LEEN*, the total data transferred in the shuffle phase is reduced by 49 % (from 15.6 GB to 7.9 GB) for the test set (6VMs 2) and reduced by 10/5 (from 21 GB to 19 GB) for the test set (30VMs).

5.3 Data distribution of reducers' inputs in *LEEN*

We compare the data distribution of reducers' inputs in both native Hadoop and *LEEN*. We use two metrics to measure the balance of map tasks distribution [25]:

- The coefficient of variation:

$$cv = \frac{stdev}{mean} \times 100\ \% \tag{9}$$

- The max-min ratio:

$$Min-Max\ Ratio = \frac{\min_{1\leq i\leq n} ReduceInput_i}{\max_{1\leq j\leq n} ReduceInput_j} \times 100\ \% \tag{10}$$

Table 3 shows the variation in the data distribution of reducers' inputs. We can see that the variation is significant in native Hadoop compared to *LEEN*. For example, for the test set (6VMs 1), *LEEN* achieves 10 times better fairness in the reducers' input than native Hadoop: the co-efficient of variation is almost 73 % and the min-max ratio is 200 % in native Hadoop while they are 7 % and 20 % in *LEEN*, respectively.

5.4 Latency of the MapReduce Jobs in *LEEN*

Regarding the latency of the whole job, we observe that, in the presence of the partitioning skew, *LEEN* outperforms native Hadoop in all the test sets, with improvement of up to 45 %. Moreover the performance improvements of *LEEN* over native Hadoop varies according to the two aforementioned factors along with two another important factors which we are going to investigate in the future: computing capacity of the nodes which can affect the execution time of reduce tasks, and network latency which can affect the time to shuffle the intermediate data among the different data nodes.

Table 3 Variation of reducers' inputs amongst different nodes for *LEEN* against native Hadoop

	cv		Min-Max Ratio	
	Hadoop	*LEEN*	*Hadoop*	*LEEN*
6VMs 1	73 %	7 %	200 %	20 %
6VMs 2	23 %	13 %	100 %	33 %
30VMs	81 %	15 %	290 %	35 %

For the test set (6VM 1), *LEEN* outperforms native Hadoop by 11.3 %: Although the latency of the first two phases—map phase and shuffle phase—in native Hadoop is lower than *LEEN* (by only 9 s), which can be explained due to the advantage of the concurrent execution of the map phase and the reduce phase (it is worth to note that it was expected that native Hadoop outperforms *LEEN* for the first two phases, especially that they transfer the same amount of data, but surprisingly the latency of these two phases were almost the same, which can be explained due to the map-skew [12] and the unfairness in the shuffled data between the nodes). However, the better fairness in reducers' inputs data between nodes in *LEEN* results in balanced reduce functions executions, which in turn makes all reducers finish almost at the same time (the time taken by the best reduce function is 150 s and the time taken by the worst reduce function is 168 s). On the other hand, in native Hadoop, the skew reduce computation is very high and this results with longer execution time of the job: some nodes will be heavy loaded while other nodes are idles (the time taken by the best reduce function is 33 s and the time taken by the worst reduce function is 231 s).

For the test set (6VM 2), *LEEN* speeds up native Hadoop by 4 %: The latency of the first two phases—map phase and shuffle phase—in native Hadoop is almost the same as in *LEEN* (less by only 14 s), which can be explained due to higher locality in *LEEN* and thus the smaller

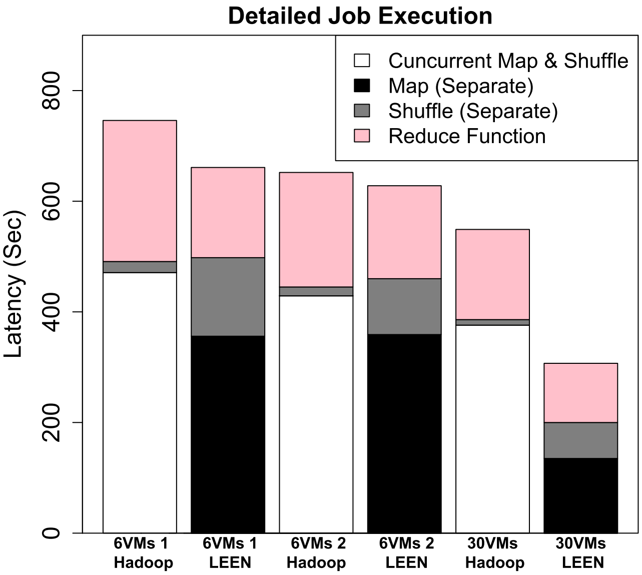


Fig. 8 Detailed performance of each stage in *LEEN* against native Hadoop for the three test sets

transferred shuffled data. Similar to test set 1, the better fairness in reducers' inputs data between nodes in *LEEN* results in balanced reduce functions executions and thus lower latency.

As we can see in Fig. 8, the latency in native Hadoop in test set (6VMs 2) is lower that the one in test set (6VMs 1), although they both achieve almost the same locality. This can be explained due to the better fairness in data distribution of reducers' inputs. On the other hand, the latency in *LEEN* in test set (6VMs 2) is lower that the one in test set (6VMs 1), although the fairness in data distribution of reducers' inputs is better in test set (6VMs 1). This is due to the almost 40 % reduction in data transfer that is achieved by *LEEN* in test set (6VMs 1).

For the test set (30VMs), *LEEN* outperforms native Hadoop by 45 %: LEEN achieves a higher locality than in native Hadoop and thus a smaller transferred shuffled data than native Hadoop. *LEEN* also achieves better fairness in reducers' inputs between nodes than in native Hadooop which in turn results in balanced reduce functions executions, all reducers therefore finish almost at the same time as shown in Fig. 9d (the time taken by the best reduce function is 40 s and the time taken by the worst reduce function is 55 s). On the other hand, in native Hadoop, skew reduce computation is very high as shown in Fig. 9c and this results with longer execution time of the job: some nodes will be heavy loaded while other nodes are idles (the time taken by the best reduce function is 3 s and the time taken by the worst reduce function is 150 s).

5.5 Influence on load balancing

Finally in this subsection, we compare the system load balancing in *LEEN* against native Hadoop. As we stated earlier, *LEEN* is designed to mitigate the reduce computation skew through fair distribution of data among reducers: *LEEN* reduces the reduce computations variation by almost 85 % compared to native Hadoop (from 90 % to 13 %). This results with balancing the load between reducers and lower latency in contrast to native Hadoop as shown in Fig. 8c and d.

As shown in Fig. 8a, in native Hadoop, even though all map tasks receive the same amount of data, the slowest task takes more than 170 s while the fastest one completes in 16 s. However, in *LEEN* the executions of map tasks vary only by 19 % as shown in Fig. 8b: the slowest task takes

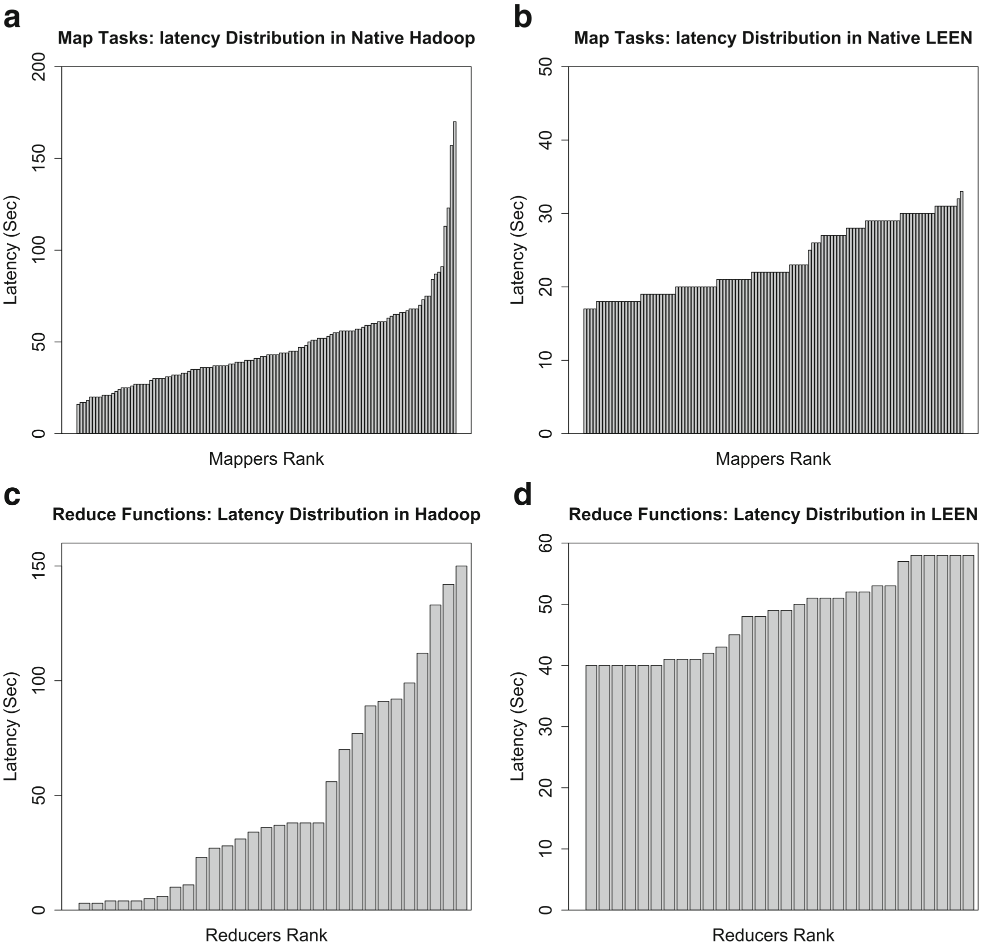


Fig. 9 Load balancing: distribution of the tasks run time for both map tasks and reduce computation for the test set 30VMs. **a** Running time of map tasks in native Hadoop. **b** Running time of map tasks in *LEEN*. **c** Running time of reduce computations in native Hadoop. **d** Running time of reduce computations in *LEEN*

more than 58 s while the fastest one completes in 40 s. This is because of the asynchronous map and reduce scheme: we start the shuffle phase after all maps are completed so here the complete I/O disk resources will be reserved to the map tasks, while in native Hadoop map tasks and reduce tasks will compete for the disk resources and this varies according the distribution of the keys during partitioning as well.

It is important to mention that this load balancing in *LEEN* comes at the cost of fully resource utilization: the network resources are not used during map phase and the cpu usage is not utilized during the shuffle phase. We are going to investigate some techniques to overlap the map and the shuffle phase while preserving the same keys design in *LEEN* in the future.

6 Related work

MapReduce has attracted much attention in the past few years. Some research has been dedicated to adopting MapReduce in different environments such as multi-core [6], graphics processors (GPU)s [5], and virtual machines [26, 27]. Many works on improving MapReduce performance has been introduced through locality-execution in the map phase [28, 29], tuning the schedulers at OS-kernel [30]. Many case studies have demonstrated efficient usage of MapReduce for many applications including scientific applications [16, 31–33], machine learning applications [20, 34] and graph analysis [35, 36].

There have been few studies on minimizing the network congestion by data-aware reduction. Sangwon et al. have proposed pre-fetching and pre-shuffling schemes for shared MapReduce computation environments [37]. While the pre-fetching scheme exploits data locality by assigning the tasks to the nearest node to blocks, the pre-shuffling scheme significantly reduces the network overhead required to shuffle key-value pairs. Like *LEEN*, the pre-shuffling scheme tries to provide data-aware partitioning over the intermediate data, by looking over the input splits before the map phase begins and predicts the target reducer where the key-value pairs of the intermediate output are partitioned into a local node, thus, the expected data are assigned to a map task near the future reducer before the execution of the mapper. *LEEN* has a different approach: By separating the map and reduce phase and by completely scanning the keys' frequencies table generating after map tasks, *LEEN* partitions the keys to achieve the best locality while guaranteeing near optimal balanced reducers' inputs. Chen et al. have proposed Locality Aware Reduce Scheduling (LARS), which de-signed specifically to minimize the data transfer in their proposed grid-enabled MapReduce framework, called USSOP [38]. However, USSOP, due to the heterogeneity of grid nodes in terms of computation power, varies the data size of map tasks, thus, assigning map tasks associated with different data size to the workers according to their computation capacity. Obviously, this will cause a variation in the map outputs. Master node will defer the assignment of reduces to the grid nodes until all maps are done and then using LARS algorithm, that is, nodes with largest region size will be assigned reduces (all the intermediate data are hashed and stored as regions, one region may contain different keys). Thus, LARS avoids transferring large regions out. Despite that *LEEN* and LARS are targeting different environments, a key difference between *LEEN* and LARS is that *LEEN* provides nearly optimal locality on intermediate data along with balancing reducers' computation in homogenous MapReduce systems.

Unfortunately, the current MapReduce implementations have overlooked the skew issue [15], which is a big challenge to achieve successful scale-up in a parallel query systems [39]. However, few studies have reported on the data skew impacts on MapReduce-based systems [13, 40]. Qiu et al. have reported on the skew problems in some bioinformatics applications [13], and have discussed potential solutions towards the skew problems through implementing those applications using Cloud technologies. Lin analyzed the skewed running time of MapReduce tasks, maps and reduces, caused by the Zipfian distribution of the input and intermediate data, respectively [14].

Recent studies have proposed solutions to mitigate the skew problem MapReduce [41–43]. Gufler et al. have proposed to mitigate reduce-skew by scheduling the keys to the reduce tasks based on cost model. Their solution uses TopCluster to capture the data skew in MapReduce and accordingly identifies its most relevant subset for cost estimation. *LEEN* approaches the same problem, which is computation skew among different reducers caused by the unfair distribution of reduces' inputs, but *LEEN* also aims at reducing the network congestion by improving the locality of reducers' inputs. Kwon et al. have proposed SkewReduce, to overcome the computation skew in MapReduce-based system where the running time of different partitions depends on the input size as well as the data values [42, 44]. At the heart of SkewReduce, an optimizer is parameterized by user-defined cost function to determine how best to partition the input data to minimize computational skew. In later work, Kown et al. have proposed SkewTune [44]. SkewTune is a system that dynamically mitigates skew which results from both: the uneven distribution of data and also uneven cost of data processing. *LEEN* approaches the same problem, which is computation skew among different reducers caused by the unfair distribution of reduces' inputs, while assuming all values have the same size, and keeping in mind reduce the network congestion by improving the locality of reducers' inputs. However, extending *LEEN* to the case when different values vary in size is ongoing work in our group.

7 Conclusions

Locality and fairness in data partitioning is an important performance factor for MapReduce. In this paper, we have developed an algorithm named *LEEN* for locality-aware and fairness-aware key partitioning to save the network bandwidth dissipation during the shuffle phase of MapReduce caused by partitioning skew for some applications. *LEEN* is effective in improving the data locality of the MapReduce execution efficiency by the asynchronous map and reduce scheme, with a full control on the keys distribution among different data nodes. *LEEN* keeps track of the frequencies of buffered keys hosted by each data node. *LEEN* achieves both fair data distribution and performance under moderate and large keys' frequencies variations. To quantify the data distribution and performance of *LEEN*, we conduct a comprehensive performance evaluation study using Hadoop-0.21.0 with and without *LEEN* support. Our experimental results demonstrate that *LEEN* efficiently achieves higher locality, and balances data distribution after the shuffle phase. As a result, *LEEN* outperforms the native Hadoop by up to 45 % in overall performance for different applications in the Cloud.

In considering future work, we are interested in adopting *LEEN* to the query optimization techniques [45, 46] for query-level load balancing and fairness. As a long-term agenda, we are interested in providing a comprehensive study on the monetary cost of *LEEN* in contrast with Hadoop considering different pricing schemes (for example the pay-as-you-go scheme and the pay-as-you-consume scheme [47]), knowing that *LEEN* always guarantees resource load balancing at the cost of concurrent resource access.

Acknowledgments This work is supported by NSF of China under grant No.61232008, 61133008 and 61073024, the National 863 Hi-Tech Research and Development Program under grant 2013AA01A213, the Outstanding Youth Foundation of Hubei Province under Grant No.2011CDA086, the National Science & Technology Pillar Program of China under grant No.2012BAH14F02, the Inter-disciplinary Strategic Competitive Fund of Nanyang Technological University 2011 No.M58020029, and the ANR MapReduce grant (ANR-10-SEGI-001). This work was done in the context of the Héméra INRIA Large Wingspan-Project (see http://www.grid5000.fr/mediawiki/index.php/Hemera).

References

1. Dean J, Ghemawat S (2008) Mapreduce: simplified data processing on large clusters. Commun ACM 51:107–113
2. Jin H, Ibrahim S, Bell T, Qi L, Cao H, Wu S, Shi X (2010) Tools and technologies for building the clouds. Cloud computing: principles systems and applications, Computer Communications and Networks. Springer-Verlag, Berlin, pp 3–20
3. Jin H, Ibrahim S, Qi L, Cao H, Wu S, Shi X (2011) The mapreduce programming model and implementations. In: Buyya R, Broberg J, Goscinski A (eds) Cloud computing: principles and paradigms. Wiley, Hoboken, pp 373–390
4. Isard M, Budiu M, Yu Y, Birrell A, Fetterly D (2007) Dryad: distributed data-parallel programs from sequential building blocks. In: Proceedings of the 2nd ACM European conference on computer systems (EuroSys '07), Lisbon, pp 59–72
5. He B, Fang W, Luo Q, Govindaraju NK, Wang T (2008) Mars: a mapreduce framework on graphics processors. In: Proceedings of the 17th international conference on parallel architectures and compilation techniques, Toronto, pp 260–269
6. Ranger C, Raghuraman R, Penmetsa A, Bradski G, Kozyrakis C (2007) Evaluating mapreduce for multi-core and multiprocessor systems. In: Proceedings of the 2007 IEEE 13th international symposium on high performance computer architecture (HPCA-13), Phoenix, pp 13–24
7. Hadoop project (2011) http://lucene.apache.org/hadoop
8. Yahoo! (2011) Yahoo! developer network, http://developer.yahoo.com/blogs/hadoop/2008/02/yahoo-worldslargest-production-hadoop.html
9. Hadoop (2011) Applications powered by hadoop: http://wiki.apache.org/hadoop/PoweredB
10. Ghemawat S, Gobioff H, Leung S-T (2003) The Google file system. SIGOPS - Oper Syst Rev 37(5):29–43
11. Condie T, Conway N, Alvaro P, Hellerstein JM, Elmeleegy K, Sears R (2010) Mapreduce online. In: Proceedings of the 7th USENIX conference on networked systems design and implementation (NSDI'10), San Jose
12. Kwon Y, Balazinska M, Howe B, Rolia J (2011) A study of skew in mapreduce applications, http://nuage.cs.washington.edu/pubs/opencirrus2011.pdf
13. Qiu X, Ekanayake J, Beason S, Gunarathne T, Fox G, Barga R, Gannon D (2009) Cloud technologies for bioinformatics applications. In: Proceedings of the 2nd workshop on many-task computing on grids and supercomputers (MTAGS '09)
14. Lin J (2009) The curse of zipf and limits to parallelization: a look at the stragglers problem in mapreduce. In: Proceedings of the 7th workshop on large-scale distributed systems for information retrieval (LSDS-IR'09)
15. DeWitt DJ, Stonebraker M (2008) Mapreduce: a major step backwards, http://databasecolumn.vertica.com/databaseinnovation/mapreduce-a-major-step-backwards
16. Wiley K, Connolly A, Gardner JP, Krughof S, Balazinska M, Howe B, Kwon Y, Bu Y (2011) Astronomy in the cloud: using MapReduce for image coaddition, CoRR abs/1010.1015
17. Chen R, Yang M, Weng X, Choi B, He B, Li X (2012) Improving large graph processing on partitioned graphs in the cloud. In: Proceedings of the third ACM symposium on cloud computing,

SoCC ’12, ACM, New York, pp 3:1–3:13. doi:10.1145/2391229.2391232. http://doi.acm.org/10.1145/2391229.2391232

18. Schatz MC (2009) Cloudburst: highly sensitive read mapping with mapreduce. Bioinformatics 25:1363–1369
19. Verma A, Llorà X, Goldberg DE, Campbell RH (2009) Scaling genetic algorithms using MapReduce. In: Proceedings of the 2009 9th international conference on intelligent systems design and applications, pp 13–18
20. Ng AY, Bradski G, Chu C-T, Olukotun K, Kim SK, Lin Y-A, Yu Y (2006) MapReduce for machine learning on multicore. In: Proceedings of the twentieth annual conference on neural information processing systems (NIPS ’06), Vancouver, pp 281–288
21. Lin J, Schatz M (2010) Design patterns for efficient graph algorithms in mapreduce. In: Proceedings of the eighth workshop on mining and learning with graphs, Washington, pp 78–85
22. Xen hypervisor homepage (2011) http://www.xen.org/
23. Amazon elastic compute cloud (2011) http://aws.amazon.com/ec2/
24. Ibrahim S, Jin H, Lu L, Wu S, He B, Qi L (2010) Leen: locality/fairness-aware key partitioning for mapreduce in the cloud. In: Proceedings of the 2010 IEEE second international conference on cloud computing technology and science (CLOUDCOM’10), Indianapolis, pp 17–24
25. Jain R, Chiu D-M, Hawe W (1984) A quantitative measure of fairness and discrimination for resource allocation in shared computer systems, DEC Research Report TR-301
26. Ibrahim S, Jin H, Lu L, Qi L, Wu S, Shi X (2009) Evaluating mapreduce on virtual machines: the hadoop case. In: Proceedings of the 1st international conference on cloud computing (CLOUDCOM’09), Beijing, pp 519–528
27. Ibrahim S, Jin H, Cheng B, Cao H, Wu S, Qi L (2009) Cloudlet: towards mapreduce implementation on virtual machines. In: Proceedings of the 18th ACM international symposium on high performance distributed computing (HPDC-18), Garching, pp 65–66
28. Zaharia M, Borthakur D, Sen Sarma J, Elmeleegy K, Shenker S, Stoica I (2010) Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th ACM European conference on computer systems (EuroSys’10), Paris, pp 265–278
29. Ibrahim S, Jin H, Lu L, He B, Antoniu G, Wu S (2012) Maestro: replica-aware map scheduling for mapreduce. In: Proceedings of the 12th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGrid 2012), Ottawa
30. Ibrahim S, Jin H, Lu L, He B, Wu S (2011) Adaptive disk i/o scheduling for mapreduce in virtualized environment. In: Proceedings of the 2011 international conference on parallel processing (ICPP’11), Taipei, pp 335–344
31. Menon RK, Bhat GP, Schatz MC (2011) Rapid parallel genome indexing with MapReduce. In: Proceedings of the 2nd international workshop on MapReduce and its applications, San Jose, pp 51–58
32. Ekanayake J, Pallickara S, Fox G (2008) Mapreduce for data intensive scientific analyses. In: Proceedings of the 2008 fourth IEEE international conference on eScience, pp 277–284
33. Gunarathne T, Wu T-L, Qiu J, Fox G (2010) MapReduce in the clouds for science. In: Proceedings of the 2010 IEEE second international conference on cloud computing technology and science, pp 565–572
34. Ganjisaffar Y, Debeauvais T, Javanmardi S, Caruana R, Lopes CV (2011) Distributed tuning of machine learning algorithms using MapReduce clusters. In: Proceedings of the 3rd workshop on large scale data mining: theory and applications, San Diego, pp 2:1–2:8
35. Blanas S, Patel JM, Ercegovac V, Rao J, Shekita EJ, Tian Y (2010) A comparison of join algorithms for log processing in mapreduce. In: Proceedings of the 2010 international conference on Management of data, Indianapolis, pp 975–986
36. Logothetis D, Trezzo C, Webb KC, Yocum K (2011) In-situ mapreduce for log processing. In: Proceedings of the 2011 USENIX conference on USENIX annual technical conference, Portland, pp 9–9
37. Seo S, Jang I, Woo K, Kim I, Kim J-S, Maeng S (2009) Hpmr: prefetching and pre-shuffling in shared mapreduce computation environment. In: Proceedings of the 2009 IEEE international conference on cluster computing (CLUSTER’09), New Orleans
38. Su Y-L, Chen P-C, Chang J-B, Shieh C-K (2011) Variable-sized map and locality-aware reduce on public-resource grids. Futur Gener Comput Syst 27(6):843–849
39. DeWitt D, Gray J (1992) Parallel database systems: the future of high performance database systems. Commun ACM 35:85–98
40. Chen S, Schlosser SW (2008) Map-reduce meets wider varieties of applications, Tech. Rep. IRP-TR-08-05, Technical Report. Intel Research Pittsburgh
41. Ananthanarayanan G, Kandula S, Greenberg A, Stoica I, Lu Y, Saha B, Harris E (2010) Reining in the outliers in map-reduce clusters using mantri. In: Proceedings of the 9th USENIX conference on operating systems design and implementation (OSDI’10), Vancouver, pp 1–16
42. Kwon Y, Balazinska M, Howe B, Rolia J (2010) Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In: Proceedings of the 1st ACM symposium on Cloud computing (SoCC ’10)
43. Gufler B, Augsten N, Reiser A, Kemper A (2012) Load balancing in mapreduce based on scalable cardinality estimates. In: Proceedings of the 2012 IEEE 28th international conference on data engineering (ICDE ’12)
44. Kwon Y, Balazinska M, Howe B, Rolia J (2012) Skewtune: mitigating skew in mapreduce applications. In: Proceedings of the 2012 ACM SIGMOD international conference on management of data (SIGMOD ’12)
45. He B, Yang M, Guo Z, Chen R, Lin W, Su B, Wang H, Zhou L (2009) Wave computing in the cloud. In: Proceedings of the 12th conference on hot topics in operating systems (HotOS’09)
46. He B, Yang M, Guo Z, Chen R, Su B, Lin W, Zhou L (2010) Comet: batched stream processing for data intensive distributed computing. In: Proceedings of the 1st ACM symposium on Cloud computing (SoCC ’10)
47. Ibrahim S, He B, Jin H (2011) Towards pay-as-you-consume cloud computing. In: Proceedings of the 2011 IEEE international conference on services computing (SCC’11), Washington, DC, pp 370–377

Shadi Ibrahim is a postdoctoral researcher within the KerData team at INRIA Rennes, working on scalable distributed data processing in the Cloud. He holds a Ph.D. in Computer Science from Huazhong University of Science and Technology in Wuhan of China. He has several years of experience with academic research at Huazhong University of Science and Technology, INRIA research center and Microsoft research center in Asia. His research interests are in the areas of cloud computing, data-intensive computing, virtualization technology and file and storage systems.

Hai Jin is a Cheung Kung Scholars Chair Professor of computer science and engineering at Huazhong University of Science and Technology (HUST) in China. He is now Dean of the School of Computer Science and Technology at HUST. Jin received his PhD in computer engineering from HUST in 1994. In 1996, he was awarded a German Academic Exchange Service fellowship to visit the Technical University of Chemnitz in Germany. Jin worked at The University of Hong Kong between 1998 and 2000, and as a visiting scholar at the University of Southern California between 1999 and 2000. He was awarded Excellent Youth Award from the National Science Foundation of China in 2001. Jin is the chief scientist of ChinaGrid, the largest grid computing project in China, and the chief scientist of National 973 Basic Research Program Project of Virtualization Technology of Computing System. Jin is a senior member of the IEEE and a member of the ACM. Jin is the member of Grid Forum Steering Group (GFSG). He has co-authored 15 books and published over 400 research papers. His research interests include computer architecture, virtualization technology, cluster computing and grid computing, peer-to-peer computing, network storage, and network security.

Lu Lu received his Bachelor degree in computer science and technology from Naval University of Engineering (China) in 2008. He is now a PhD candidate student in Service Computing Technolo gy and System Lab (SCTS) and Cluster and Grid Lab (CG CL) at Huazhong University of Science and Technology (China). His research interests focus on distributed data processing and graph computing.

Bingsheng He received the bachelor degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science in Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in Division of Networks and Distributed Systems, School of Computer Engineering of Nanyang Technological University, Singapore. His research interests are high performance computing, cloud computing, and database systems.

Gabriel Antoniu is a senior research scientist at INRIA. He is leading of the KerData research team at INRIA Rennes - Bretagne Atlantique. He received his Ph.D. degree in Computer Science in 2001 from ENS Lyon and his Habilitation for Research Supervision (HDR) from ENS Cachan in 2009. His research interests include: big data, grid and cloud storage, large-scale distributed data management and sharing, data consistency models and protocols, grid and peer-to-peer systems. In the area of distributed data storage, he is leading the MapReduce ANR project (2010- 2013) in partnership with Argonne National Nab (USA), University of Illinois at Urbana Champaign (USA), and INRIA-UIUC Joint Lab for Petascale Computing and IBM France. He is co-leading the AzureBrain INRIA-Microsoft Research project (2010-2013) on optimized cloud storage for joint genetics and neuroimagiong analysis.

Song Wu is a Professor of Computer Science and Engineering at Huazhong University of Science and Technology (HUST). He received his Ph.D. from HUST in 2003. He is now the head of Parallel and Distributed Computing Institute of HUST,and also serves as the vice director of Service Computing Technology and System Lab (SCTS) and Cluster and Grid Computing Lab (CGCL) of HUST. His research interests are in the areas of cloud computing, virtualization technology and system architecture.