2nd IEEE International Conference on Cloud Computing Technology and Science

LEEN: Locality/Fairness- Aware Key Partitioning for MapReduce in the Cloud

Shadi Ibrahim, Hai Jin, Lu Lu, Song Wu
Cluster and Grid Computing Lab
Services Computing Technology and System Lab
Huazhong University of Science and Technology
Wuhan, 430074, China
{shadi, hjin, wusong}@hust.edu.cn

Bingsheng He
School of Computer Engineering
Nanyang Technological University
Singapore, 639798
bshe@ntu.edu.sg

Li Qi
Information and Technology Department
China Development Bank
Beijing, 100037, China
quick.qi@gmail.com

***Abstract*—This paper investigates the problem of *Partitioning Skew*[1] in MapReduce-based system. Our studies with Hadoop, a widely used MapReduce implementation, demonstrate that the presence of partitioning skew causes a 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 experience performance degradation due to the long data transfer during the shuffle phase along with the computation skew, particularly in reduce phase. 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-0.18.0. 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 40% on different workloads.**

Keywords-MapReduce; Hadoop; partationing skew; Cloud Computing

I. 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 platform. 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. Many implementations have been developed in different programming languages for various purposes [2-5]. The popular open source implementation of MapReduce, Hadoop [5], was developed primarily by Yahoo, where it processes hundreds of terabytes of data on tens of thousands of nodes [6], and is now used by other companies, including Facebook, Amazon, Last.fm, and the New York Times [7].

The MapReduce system runs on top of the *Google File System* (*GFS*) [8], 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 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 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-key) Modulo ReduceID)

The hashing results are stored in memory buffers, before spilling the intermediate data (index file and data file) to the local disk [9]. 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 with 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, we show that, 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 execution in MapReduce system is not enough), (2) unfairness of reducers' inputs, and

[1] We refer to the significant variance in both intermediate keys' frequencies and their distributions among the different data nodes as ***Partitioning Skew***.

finally (3) severe performance degradation (i.e. the variance of reducers' inputs, in turn, causes a variation in the execution time of reduce tasks, resulting with longer response time of the whole job, as the job's response time is dominated by the slowest reduce instance).

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 with 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 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 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 move 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 0.18.0. 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 40%.

We summarize the contributions of our paper as follows:

- 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 discusses the related works. Section 3 illustrates the recent partitioning strategy used in Hadoop. The design of the asynchronous map and reduce scheme and the *LEEN* scheduling algorithm is discussed in section 4. Section 5 details the performance evaluation. Finally, we conclude the paper and propose our future work in section 6.

II. RELATED WORKS

We divide the previous work into two groups: (1) reduce the network congestion by data-aware shuffling, and (2) the impacts of data skew on MapReduce performance.

A. *Data-aware Reduce Execution*

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 environment [10]. 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 designed specifically to minimize the data transfer in their proposed grid-enabled MapReduce framework, called USSOP [11]. 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 system.

B. *Data Skew in MapReduce*

Unfortunately, the current MapReduce implementations have overlooked the skew issue [12], which is a big challenge to achieve successful scale-up in parallel query systems [13].

However, few studies have reported on the data skew impacts on MapReduce-based system. Qiu et al. have reported on the skew problems in some bioinformatics applications [14], 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 [15].

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 [16]. 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. *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.

III. Empirical Study on the Impact of Partationing Skew in MapReduce

To justify our motivation, we demonstrate the problem by following motivational example and by performing a series of experiments to demonstrate the aforementioned problems in the current Hadoop implementation.

A. *Motivational Example*

As shown in Fig. 1, three nodes: node1, node2, and node3, with nine intermediate keys, are ordered by their appearance during the map tasks execution. The sum of the entire keys' frequencies is 225 keys, distributed equally among the three identical data nodes, we assume balanced execution of the map tasks as well as we assume that the size of all intermediate records is equal (extending our work to the case of different records' size, when the values associated with the keys are differ in size, is ongoing research within our group). In our example the keys' frequencies are varied along with their distributions among the data nodes.

Fig. 1 illustrates the keys partitioning results using the existing hash partitioning. We observe that the current blind hash partitioning is inadequate in the case of *partitioning skew* in terms of data size which needs to be shuffled through the system network and balanced distribution of the reduces' inputs. In our example, the percentages of the keys locally partitioned on each of the three nodes, (*Locality = Local keys/Total Map Output*), are 29%, 40%, and 22%, respectively, with an average of 30%. Subsequently, the total keys needs to be transferred (*Total Map output × (1 − Locality)*) is 156 keys which is big fraction of the maps output. Moreover, the reducers' input varies by 42%. The reducers' inputs are: 82, 102, and, 41.

In summary, the absence of locality-aware keys partitioning overlooks any opportunities to reduce the data transfer during shuffle phase. In addition, the imbalanced data distribution of reducers' inputs among the different data node occurs; consequently, heavy reduce execution on some nodes (node1 and node2). Thus performance experiences degradation (i.e. waiting the last subtask to be finished), and less resource utilization (i.e. node3 will be idle while node2 is overloaded).

B. *Empirical Study*

1) *Experimental environment*

Our experimental hardware consists of a cluster with seven nodes. Each node is equipped with two quad-core 2.33GHz Xeon processors, 8GB of memory and 1TB of disk, runs RHEL5 with kernel 2.6.22, and is connected with 1GB Ethernet. All results described in this paper are obtained using Hadoop version 0.18.0. To monitor the data flow in our experiments we use iptables [17], that is, a TCP/IP packet filtering system embedded into Linux kernel. We perform our experiments by repeatedly executing the wordcount benchmark, with dataset of 2GB.

In order to show the case of partitioning skew, we perform the wordcount applications without combiner function. Moreover, we have used up to 1000 different keys, representing different words with the same length (to avoid variation in values size), with different frequencies.

2) *Key Results*

Fig. 2 presents the data locality, the data transferred and the data distribution. We observe that the data locality ranges from 1% to 41% among the different data nodes, in particular, 14%, 9%, 4%, 17%, 1%, and 41%, with average of 17%. Moreover, the total data shuffled is 2966MB which is greater than the input data (2GB = 2048MB).

Furthermore, the data distribution among the different data nodes is totally imbalanced, ranged from 182MB to 903MB. We use two metrics [18] to illustrate the imbalance in data distribution, shown in Table 1.

- The coefficient of variation:

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

- The max-min ratio:

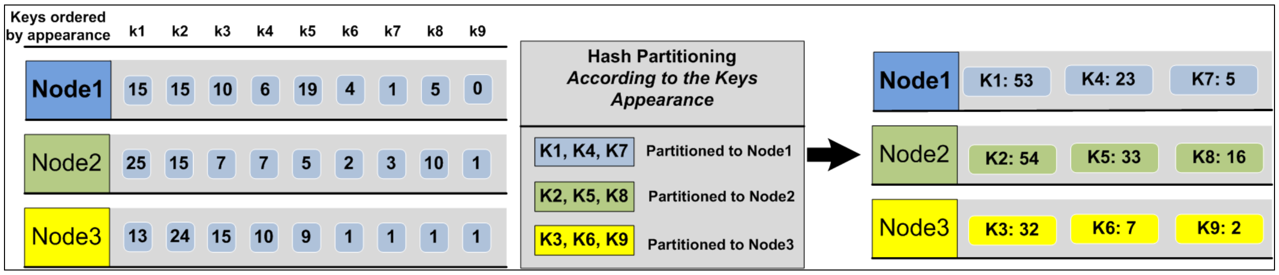

Keys ordered by appearance
k1 k2 k3 k4 k5 k6 k7 k8 k9
Node1 15 15 10 6 19 4 1 5 0
Node2 25 15 7 7 5 2 3 10 1
Node3 13 24 15 10 9 1 1 1 1
Hash Partitioning According to the Keys Appearance
K1, K4, K7 Partitioned to Node1
K2, K5, K8 Partitioned to Node2
K3, K6, K9 Partitioned to Node3
Node1 K1: 53 K4: 23 K7: 5
Node2 K2: 54 K5: 33 K8: 16
Node3 K3: 32 K6: 7 K9: 2

Figure 1. Motivational Example: *demonstrates 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.*

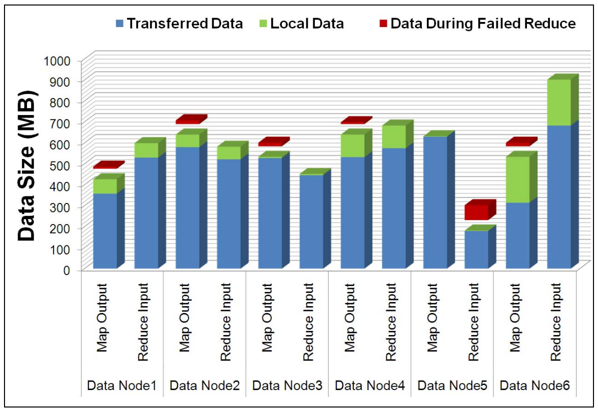


Figure 2. The size of data transferred from and into the data nodes during the copy phase, when performing wordcount application on 2GB of data after disabling the combiners.

$$Max\text{-}Min\ Ratio = \frac{\min_{1\le i\le n}(Reduce\ input_i)}{\max_{1\le j\le n}(Reduce\ input_j)} \times 100\%$$

TABLE I. VARIATION OF THE DATA DISTRIBUTION AMONG DIFFERENT DATA NODES.

	Data Distribution
Max-Min Ratio	20%
cv	42%

As presented in Table 1, the variation among different data nodes in term of data distribution is 42%. While the minimum data distribution, data node5 in our example host data set of 20% compared to the maximum data node, data node6. Resulting with misuse of the system resources, for example data node5 finished reading the input data and process the reduce function nearly five times faster than data node6.

IV. *LEEN* DESIGN

In this section we first introduce the asynchronous map and reduce scheme. Then, we present *LEEN* algorithm.

A. *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 key's distributions, we propose to use asynchronous map and reduce scheme, which is 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 seems 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 is 26 seconds while it is 32 seconds in the native Hadoop. Moreover, the speedup of map execution 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 VM with small memory size (e.g. in Amazon EC2 [19], the small instance has 1GB 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 the a unique ID which is the KeyID: $hash(Hash\,code\,(Intermediate\text{-}key)\ Modulo\ KeyID)$

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 includes the number of the records in each file, representing the key frequency. Finally, when all the maps done all the metadata files will be aggregated by the *Job Tracker*, the keys will be partitioned to the different data nodes according to *LEEN* algorithm.

B. *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 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 (*Fairness/Locality*).

Locality is the sum of keys frequencies in the nodes, which are partitioned to, to the total keys frequencies.

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

where FK_i^j indicate the frequency of key k_i in the data node rn_j. If k_i is partitioned to n_j, 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: $FK_i = \sum_{j=1}^{nodes} FK_i^j$. The locality in our system will be bounded by:

$$\frac{\sum_{i=1}^{K} min_{1\le j\le n}(FK_i^j)}{\sum_{i=1}^{K} FK_i} < Locality_{LEEN} < \frac{\sum_{i=1}^{K} max_{1\le j\le n}(FK_i^j)}{\sum_{i=1}^{K} FK_i}$$

Fairness is the variation of the reducers' inputs. In

MapReduce system 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 $D_{overload}$:

$$D_{overload} = max\,(Reducers\ input) - Mean$$
$$= max\,(HostedDataN_K^j) - \frac{Total\ Data}{N}$$

where $HostedDataN_i^j$ is the data hosted in node n_j after partitioning all the K keys.

$$HostedDataN_i^j = \begin{cases} SumKN^j & ,the\ intial\ value \\ HostedDataN_{i-1}^j + (FK_i - FK_i^j) & ,k_i\ is\ partationed\ to\ n_j \\ HostedDataN_{i-1}^j - FK_i^j & ,k_i\ is\ not\ partationed\ to\ n_j \end{cases}$$

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 keys are sorted. Thus we sort the keys according to their (*Fairness/Locality*) values. As keys with small value will have less impact on the global (*Fairness/Locality*), therefore, we sort the keys in descending order according to their fairness-locality value, refereed as *FLK*.

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

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

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

where FK_i^j indicates the frequency of key k_i in the data node n_j, and *Mean* represents the mean of FK_i^j values. The best locality indicates partitioning k_i *to the* data node n_j which has the maximum frequencies. FLK_i can be formulated as:

$$FLK_i = \frac{DevK_i}{\max_{1\le j\le n} FK_i^j}$$

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 (*total data/the number of data nodes*).

For a specific key 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:

$$Fairness-ScoreN_i^j = \sqrt{\frac{\sum_{j=1}^{n}(HostedDataN_i^j - Mean)^2}{N}}$$

It is very important that our heuristic method has running time at most $K \times N$. The complete algorithm is represented in Fig. 3. Fig. 4 demonstrates the results of using *LEEN* with the same motivated example mentioned in section 3.1.

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 FK_i^j values.
Output: *partition* (k_i, n_j)

```
foreach $k_i \in K$ do
// process the nodes according to their $FK_i^j$
    j← 0
    while $Fairness-ScoreN_i^j > Fairness-ScoreN_i^{j+1}$ do
    j←j+1
    end while
    Partition $(k_i,n_j)$
    foreach $n_j \in N$ do
        Calculate $HostedDataN_i^j$
    end for
end for
```

Figure 3. LEEN Algorithm

V. Performance Evaluation

A. Evaluation Methodology

The testbed is similar to the one descried in section 3.2.1. In order to extend our testbed, we also use virtualized environment, using Xen [20]. In the virtualized environment, we deploy four *virtual machines* (VM) on each *physical machine* (PM), reaching a cluster size of 24 data nodes. Each virtual machine is configured with 1 CPU and 1GB memory. We conduct our experiments with native Hadoop-0.18.0 and then with *LEEN*.

To evaluate the performance of *LEEN*, we have designed an execution framework to execute *LEEN*. The framework consists of three separate stage: (1) map execution: execute all the maps and store the intermediate data files in the local disk and perform a merge function on the files with the same keyID, side by side with generating the keys' frequencies table; (2) use *LEEN* algorithm to partition the keys to their appropriate data nodes; and (3) reduce execution: start the shuffle phase and then perform sort and reduce function on the reduce input. The sum of the three stages represents the execution time of the whole job. We use wordcount benchmark without combiner.

In our experiments using the keys' frequencies variation and the key's distribution are very important parameters in the motivation of *LEEN* design. While, the keys' frequencies variation will obviously cause variation of the data distribution of reducers' inputs, the variation in key's distribution will affect the amount of data transferred during 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, and we get six different test sets shown in Table 2. Moreover, Table 2 shows the locality boundaries could be achieved in each test set which can be calculated by the $Locality_{LEEN}$ boundaries defined in section 4.2.

B. *Key Results*

In order to investigate the performance impacts of partationing skew, caused by the wide variety of both intermediate keys' frequencies and their distribution among the different datanodes, we first study the impacts of the two factors separately: test sets #1 and #2 shown in Table 2.

In the test set #1, we vary the keys frequencies, while we keep uniform distribution of each key among the data nodes. As expected, shown in Fig. 5-a both *LEEN* and native Hadoop achieve close to maximum possible locality (presented in Table 2), resulting with minimum data transfer with both. Thus, native Hadoop outperforms *LEEN* in the sum of the first two phases, map phase and shuffle phase, which can be explained due to advantage of the concurrent execution of map phase and shuffle phase in native Hadoop over *LEEN*. However, due to the keys' frequencies variation, *LEEN* achieves 10 times better fairness in the reducers' input than native Hadoop (see Fig. 5-b), resulting with faster execution of the reduce tasks and overall performance improvement of 6%.

In the test set #2, we vary the distribution of each key among different data node, and kept the frequencies of all the keys nearly equal. Subsequently, we get the same fairness in both native Hadoop and *LEEN*. The reducers' inputs variations are less than 1% (see Fig. 5-b). Thus, *LEEN* and native Hadoop spend nearly the same time on execution the reduce tasks. However, due to the variation of key's distribution and the blindly partitioning of native Hadoop, *LEEN*, benefiting of the highly achieved locality, outperforms native Hadoop in the sum of both map phase and shuffle phase. As a result, *LEEN* outperforms native Hadoop by 9%.

Test sets #3, #4, #5, and #6, illustrate the impacts of partitioning skew while varying the two metrics, variation of keys' frequencies and variation of key's distribution, in different system configuration. In all scenarios, *LEEN* outperform native Hadoop by up to 40%.

In summary, shown in Fig. 5-a, *LEEN* achieves very high locality, proportional to the key's variation, compared with native Hadoop. Moreover, *LEEN* achieves better balance in the distribution of data among the different reduces' inputs, proportional to keys' frequencies variation, shown in Fig. 5-b. Furthermore, shown in Fig. 5-c, although, *LEEN* performs a little overhead during the map phase due to the table generation, the map phase in *LEEN* is faster than native Hadoop, which can be explained due to the concurrent execution of maps and reduce, that is, map and reduces tasks will compete for the I/O resources for reading data and spilling intermediate files in the maps and writing the data to the local disk when shuffle phase starts. Moreover, as expected *LEEN* is faster than native Hadoop during the reduce task execution, due to the improvement in the variation of the reducers' inputs shown in Fig. 5-b. Furthermore, the sums of the map and shuffle phases are depending on the achieved locality.

However, regarding the response time of the whole job, we observe that, in the presence of partitioning skew, *LEEN* outperforms native Hadoop in all the test sets, with

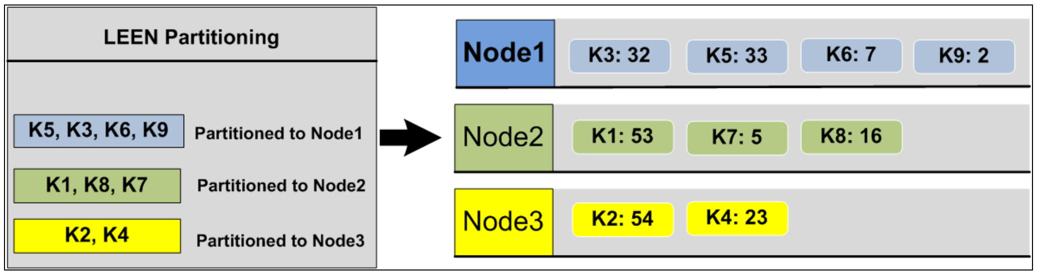
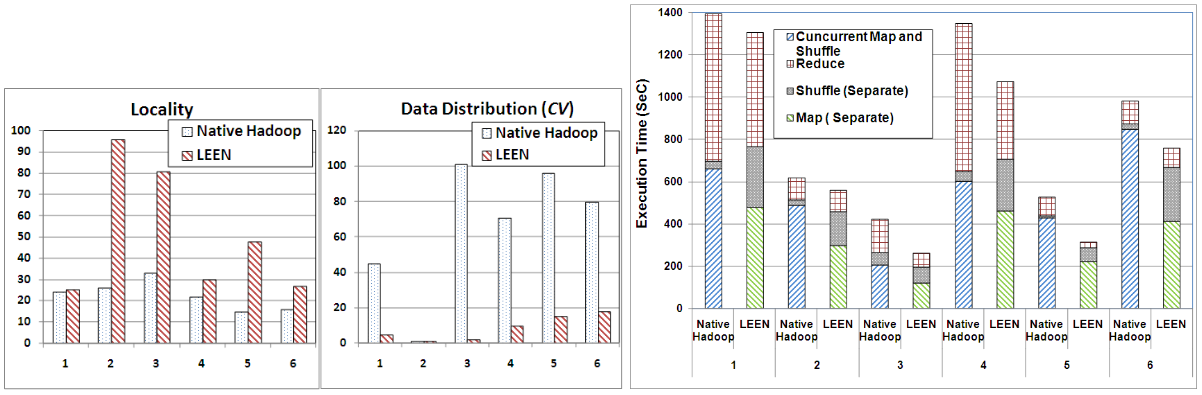


Figure 4. Motivational Example Using *LEEN*: *The keys which will be locally partitioned on the each of the three nodes are 44%, 50%, and 46% respectively, with an average of 47%, and 50% improvement of the data locality in native Hadoop as shown in Fig.1. Subsequently, LEEN reduces the amount of data transfer by 24%, 120 keys were shuffled. More importantly, LEEN achieved very close to optimal data distribution of reduces' inputs, 74, 74, and 77 respectively, and the achieved variation is 2% only.*

TABLE II. TEST SETS USED IN THE EXPERIMENTS

	1	2	3	4	5	6
Nodes number	6PMs	6PMs	6PMs	6PMs	24VMs	24VMs
Data Size	14GB	8GB	4.6GB	12.8GB	6GB	10.5GB
Keys frequencies variation	230%	1%	117%	230%	25%	85%
Key distribution variation (average)	1%	195%	150%	20%	180%	170%
Locality Range	24-26%	1-97.5%	1-85%	15-35%	1-50%	1-30%



(a) (b) (c)

Figure 5. Experiments results: (a) shows the data locality, (b) shows the data distribution variation (coefficient of variation), and (c) illustrates the execution of each phase and demonstrates the response time for the six experiments sets using native Hadoop partitioning strategy and *LEEN*.

improvement of up to 40%. Moreover the performance improvements of *LEEN* over native Hadoop varies according to the two aforementioned factors along with two another important factors which are 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, hown in test set #5 and #6.

VI. CONCLUSION AND FUTURE WORK

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-0.18.0 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 40% in overall performance for different applications in the Cloud.

Future Work. In considering future work, we are going to release the asynchronous MapReduce scheme and *LEEN* algorithm as optional plug-in in Hadoop-0.18.0 and higher versions. Moreover, we are intending to conduct more experiments to evaluate *LEEN* with different applications such as scientific applications [14, 21]. In addition to provide a comprehensive study on the impact of the system configuration on *LEEN* performance, including the impacts of virtualization technology as in [22, 23], the impacts CPU and memory capacity (as mentioned in section 4.1) and the impacts of different network topology including: star, tree and Dcell (motivated by the research work in [24] which reported on the impacts of network topology on MapReduce).

As a long-term agenda, we are interested in adopting *LEEN* to the query optimization techniques [25, 26] for query-level load balancing and fairness.

VII. ACKNOWLEDGMENTS

The research work is supported by National 973 Key Basic Research Program under grant No.2007CB310900, Information Technology Foundation of MOE and Intel under grant MOE-INTEL-09-03, the Important National Science & Technology Specific Projects under grant 2009ZX03004-002, National High-Tech R&D Plan of China under grant 2006AA01A115, and China Next Generation Internet Project under grant CNGI2008-109.

REFERENCES

[1] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters", *Proc. Usenix Symp. Opearting Systems Design & Implementation (OSDI 2004)*, Dec. 2004, pp.137-150.

[2] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks", *Proc. European Conf. Computer Systems (EuroSys 2007)*, ACM Press, Mar. 2007, pp.59-72.

[3] B. S. He, W. B. Fang, Q. Luo, N. K. Govindaraju, and T. Y. Wang, "Mars: a MapReduce framework on graphics processors", *Proc. conf. Parallel Architectures and Compilation Techniques (PACT 2008)*, ACM Press, Oct. 2008, pp.260-269.

[4] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, "Evaluating MapReduce for Multi-core and Multiprocessor Systems", *Proc. Symp. High-Performance Computer Architecture (HPCA-13)*, ACM Press, Feb. 2008, pp.13-24.

[5] Hadoop, http://lucene.apache.org/hadoop

[6] Yahoo!, Yahoo! Developer Network, http://developer.yahoo.com/blogs/hadoop/2008/02/yahoo-worlds-largest-production-hadoop.html

[7] Hadoop, Applications powered by Hadoop: http://wiki.apache.org/hadoop/PoweredB

[8] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system", *Proc. ACM Symp. Operating Systems Principles (SOSP 2003)*, ACM Press, Oct. 2003, pp.29-43.

[9] T. Condie, N. Conway, P. Alvaro, M. J. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online", *Proc. Usenix Symp. Networked Systems Design and Implementation (NSDI 2010)*, Apr. 2010. pp.313-328.

[10] S. Seo, I. Jang, K. C. Woo, I. Kim, J. S. Kim, and S. Maeng, "HPMR: Prefetching and pre-shuffling in shared MapReduce computation environment", *Proc. IEEE Conf. Cluster Computing (Cluster 2009)*, IEEE Press, Aug. 2009, pp.1-8.

[11] P. C. Chen, Y. L. Su, J. B. Chang, and C. K. Shieh, "Variable-Sized Map and Locality-Aware Reduce on Public-Resource Grids", *Proc. Conf. Grid and Pervasive Computing (GPC 2010)*, May 2010, pp.234-243.

[12] D. DeWitt and M. Stonebraker, "MapReduce: A major step backwards", http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/, 2008.

[13] J. D. DeWitt and J. Gary, "Parallel Database System: The Future of High Performance Database Systems", *Commun. ACM*, Vol.35, No.6, pp.85-98, June 1992.

[14] X. Qiu, J. Ekanayake, S. Beason, T. Gunarathne, G. Fox, R. Barga, and D. Gannon, "Cloud technologies for bioinformatics applications", *Proc. ACM Work. Many-Task Computing on Grids and Supercomputers (MTAGS 2009)*, ACM Press, Nov. 2009.

[15] J. Lin, "The Curse of Zipf and Limits to Parallelization: A Look at the Stragglers Problem in MapReduce", *Proc. Work. Large-Scale Distributed Systems for Information Retrieval (LSDS-IR'09)*, Jul. 2009.

[16] Y. C. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions", *Proc. ACM Symp. Cloud Computing (SOCC 2010)*, ACM Press, Jun. 2010, pp.75-86.

[17] Iptables: http://en.wikipedia.org/wiki/Iptables.2009

[18] R. Jain, D. Chiu, and W. Hawe, "A Quantitative Measure Of Fairness And Discrimination For Resource Allocation In Shared Computer Systems", *DEC Research Report TR-301*, September 1984.

[19] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2/

[20] Xen Hypervisor Homepage, http://www.xen.org/

[21] S. Chen and S. W. Schlosser, "Map-Reduce Meets Wider Varieties of Applications", *IRP-TR-08-05*, Technical Report, Intel Research Pittsburgh, May 2008.

[22] S. Ibrahim, H. Jin, B. Cheng, H. Cao, W. Song, and L. Qi, "Cloudlet: Towards MapReduce implementation on Virtual machines", *Proc. ACM Symp. High Performance Distributed Computing (HPDC 2009)*, ACM Press, Jun. 2009, pp.65-66.

[23] S. Ibrahim, H. Jin, L. Lu, L. Qi, S. Wu, and X. Shi, "Evaluating MapReduce on Virtual Machines: The Hadoop Case", *Proc. Conf. Cloud Computing (CloudCom 2009)*, Springer LNCS, Dec 2009, pp.519-528.

[24] G. Y. Wang, A. Butt, P. Pandey, and K. Gupta, "A Simulation Approach to Evaluating Design Decisions in MapReduce", *Proc. IEEE Symp. Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, IEEE Press, Sep. 2009, pp.1-11.

[25] B. S. He, M. Yang, Z. Y. Guo, R. S. Chen, W. Lin, B. Su, and L. D. Zhou, "Comet: Batched Stream Processing for Data Intensive Distributed Computing", *Proc. ACM Symp. Cloud Computing (SOCC 2010)*, ACM Press, Jun. 2010, pp. 63-74.

[26] B. S He, M. Yang, Z. Y. Guo, R. S. Chen, W. Lin, B. Su, H. Y. Wang, and L. D. Zhou, "Wave Computing in the Cloud", *Proc. Work. Hot Topics in Operating Systems (HotOS 2009)*, May 2009.