LIBRA: Lightweight Data Skew Mitigation in MapReduce

Qi Chen, Jinyu Yao, and Zhen Xiao, *Senior Member, IEEE*

Abstract—MapReduce is an effective tool for parallel data processing. One significant issue in practical MapReduce applications is data skew: the imbalance in the amount of data assigned to each task. This causes some tasks to take much longer to finish than others and can significantly impact performance. This paper presents LIBRA, a lightweight strategy to address the data skew problem among the reducers of MapReduce applications. Unlike previous work, LIBRA does not require any pre-run sampling of the input data or prevent the overlap between the map and the reduce stages. It uses an innovative sampling method which can achieve a highly accurate approximation to the distribution of the intermediate data by sampling only a small fraction of the intermediate data during the normal map processing. It allows the reduce tasks to start copying as soon as the chosen sample map tasks (only a small fraction of map tasks which are issued first) complete. It supports the split of large keys when application semantics permit and the total order of the output data. It considers the heterogeneity of the computing resources when balancing the load among the reduce tasks appropriately. LIBRA is applicable to a wide range of applications and is transparent to the users. We implement LIBRA in Hadoop and our experiments show that LIBRA has negligible overhead and can speed up the execution of some popular applications by up to a factor of 4.

Index Terms—MapReduce, data skew, sampling, partitioning

1 Introduction

The past decade has witnessed the explosive growth of data for processing. Large Internet companies routinely generate hundreds of tera-bytes of logs and operation records. MapReduce has proven itself to be an effective tool to process such large datasets [1]. It divides a job into multiple small tasks and assign them to a large number of nodes for parallel processing. Due to its remarkable simplicity and fault tolerance, MapReduce has been widely used in various applications, including web indexing, log analysis, data mining, scientific simulations, machine translation, etc.. There are several parallel computing frameworks that support MapReduce, such as Apache Hadoop [2], Google MapReduce [1], and Microsoft Dryad [3], of which Hadoop is open-source and widely used.

The job completion time in MapReduce depends on the slowest running task in the job. If one task takes significantly longer to finish than others (the so-called *straggler*), it can delay the progress of the entire job. Stragglers can occur due to various reasons, among which data skew is an important one. Data skew refers to the imbalance in the amount of data assigned to each task, or the imbalance in the amount of work required to process such data. The fundamental reason of data skew is that datasets in the real world are often skewed and that we do not know the distribution of the data beforehand. Note that this problem cannot be solved by the speculative execution strategy in MapReduce [4].

Data skew is not a new problem specific to MapReduce. It has been studied previously in the parallel database literature, but only limited on join [5], [6], [7], [8], [9], group [10], and aggregate [11] operations. Although some of these techniques have already been applied to MapReduce, users still need to develop their own data skew mitigation methods for specific applications in most cases. The Hadoop implementation of MapReduce by default uses static hash functions to partition the intermediate data. This works well when the data is uniformly distributed, but can perform badly when the input is skewed (some key values are significantly more frequent than others). This can be illustrated in the top figure of Fig. 1 when we run the sort benchmark [2] on 10 GB input data following the Zipf distribution ($\sigma = 1.0$). This situation also appears in other static partition methods. For example, in the bottom figure, we use a static range partition method (RADIX partition with 26 reducers for words starting with each letter of the alphabet and another reducer for special characters) to generate a lexicographically ordered inverted index on full English Wikipedia archive with a total data size of 31 GB. Like the hash method, it results in significant data skew as well. To tackle this problem, Hadoop provides a dynamic range partition method which conducts a pre-run sample of the input before the real job. The middle figure (same experiment environment as the top figure) shows that this method mitigates the problem somewhat, but the resulting distribution is still uneven.

The data skew problem in MapReduce has been studied only recently [12], [13], [14], [15], [16]. Among the solutions proposed, some are specific to a particular type of applications, some require a pre-sample of the input data, and some cannot preserve the total ordered result as the applications require. To make matters more complicated, the computing environment for MapReduce in the real world can be heterogeneous as well—multiple generations of hardware are likely to co-exist in the same data center [17]. When MapReduce runs in a virtualized cloud computing environment such as Amazon EC2 [18], the computing and

• *The authors are with the Department of Computer Science at Peking University, Beijing 100871, China. E-mail: {chenqi, yjy, xiaozhen}@net.pku.edu.cn.*

Manuscript received 26 Jan. 2014; revised 29 June 2014; accepted 15 Aug. 2014. Date of publication 21 Aug. 2014; date of current version 7 Aug. 2015. Recommended for acceptance by S. Aluru.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TPDS.2014.2350972

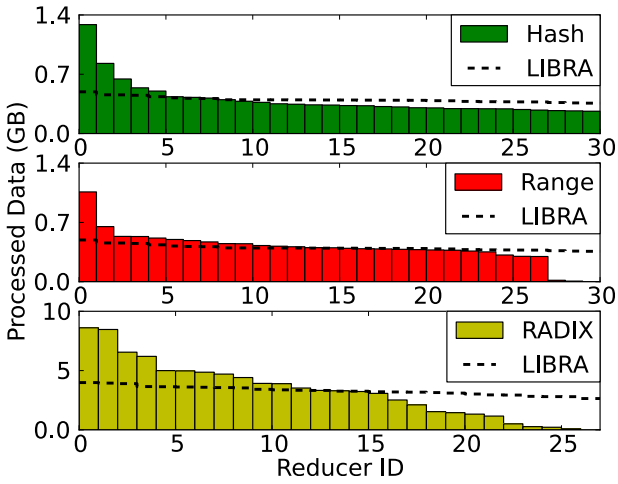


Fig. 1. Data processed by each reducer.

storage resources of the underlying virtual machines (VMs) can be diverse for a variety of reasons. A good partition method should take this into consideration instead of always dividing the work evenly among all reducers.

In this paper, we present a new strategy called LIBRA (Lightweight Implementation of Balanced Range Assignment) to solve the data skew problem for reduce-side applications in MapReduce. Compared to the previous work, our contributions include the following:

- We propose a new sampling method for general user-defined MapReduce programs. The method has a high degree of parallelism and very little overhead, which can achieve a much better approximation to the distribution of the intermediate data.
- We use an innovative approach to balance the load among the reduce tasks which supports the split of large keys when application semantics permit. Fig. 1 shows that with our LIBRA method, each reducer processes roughly the same amount of data.
- When the performance of the underlying computing platform is heterogeneous, LIBRA can adjust its workload allocation accordingly and can deliver improved performance even in the absence of data skew.
- We implement LIBRA in Hadoop and evaluate its performance for some popular applications. Experiment results show that LIBRA can improve the job execution time by up to a factor of 4.

The rest of the paper is organized as follows. Section 2 provides a background on MapReduce and the causes of data skew. Section 3 describes the implementation of our LIBRA system and Section 4 presents its algorithm details. Performance evaluation is in Section 5. Section 6 discusses related work. Section 7 concludes.

2 Background

2.1 MapReduce Framework

In a MapReduce system, a typical job execution consists of the following steps: 1) After the job is submitted to the MapReduce system, the input files are divided into multiple parts and assigned to a group of map tasks for parallel processing. 2) Each map task transforms its input (K1, V1) tuples into intermediate (K2, V2) tuples according to some user defined *map* and *combine* functions, and outputs them to the local disk. 3) Each reduce task copies its input pieces from all map tasks, sorts them into a single stream by a multiway merge, and generates the final (K3, V3) results according to some user defined *reduce* function.

In the above steps, the intermediate data generated by a map task are divided according to some user defined partitioner. For example, Hadoop uses the hash partitioner by default. Each partition is written as a continuous part of the output file. Since all map tasks use the same partitioner, all tuples that share the same key will be dispatched to the same partition. We call these tuples a *cluster*. As a result, the number of clusters is equal to the number of distinct keys in the input data. Each reduce task copies its partition (containing multiple clusters) from every map task and processes it locally.

Some applications require a total order of the output data. For example, a word count application may require the output to be in alphabetic order. Some partitioners, such as range partitioner in Hadoop, can preserve total ordering. However, the default hash partitioner does not support total ordering.

2.2 Data Skew in MapReduce

To maximize performance, ideally we want all tasks to finish around the same time. When some task takes an unusually long time to complete, it is called a *straggler* and can delay the progress of the job significantly. For stragglers caused by external factors such as faulty hardware, slow machines, etc., *speculative execution* is an effective solution where the slow task also runs on an alternative machine with the hope that it can finish there faster. Google has observed that speculative execution can decrease the job execution time by 44 percent [1]. Unfortunately, when a straggler is caused by data skew (i.e., it has to process more data than the other tasks), it cannot be solved by simply duplicating the task on another machine.

Data skew often comes from the physical properties of objects (e.g., the height of people obeys a normal distribution) and hot spots on subsets of the entire domain (e.g., the word frequency appearing on the documents obeys a Zipfian distribution). A common measurement for data skew is the *coefficient of variation*: $\frac{stddev(\vec{x})}{mean(\vec{x})}$, where $\vec{x}$ is a vector that contains the data size processed by each task. Larger coefficient indicates heavier skew.

Data skew can occur in both the map phase and the reduce phase. Map skew occurs when some input data are more difficult to process than others, but it is rare and can be easily addressed by simply splitting map tasks. Lin [19] has provided an application-specific solution that split large, expensive records into some smaller ones. In contrast, data skew in the reduce phase (also called reduce skew or partitioning skew) is much more challenging. The MapReduce framework requires that all tuples sharing the same key be dispatched to the same reducer. However, for an arbitrary MapReduce application, the distribution of the intermediate data cannot be determined ahead of time. We need to face that many real world applications exhibit large amount of data skew, including scientific applications [20], [21], distributed database operations like join, grouping and aggregation [5], [6], [8], [9], [10], [11], search engine applications

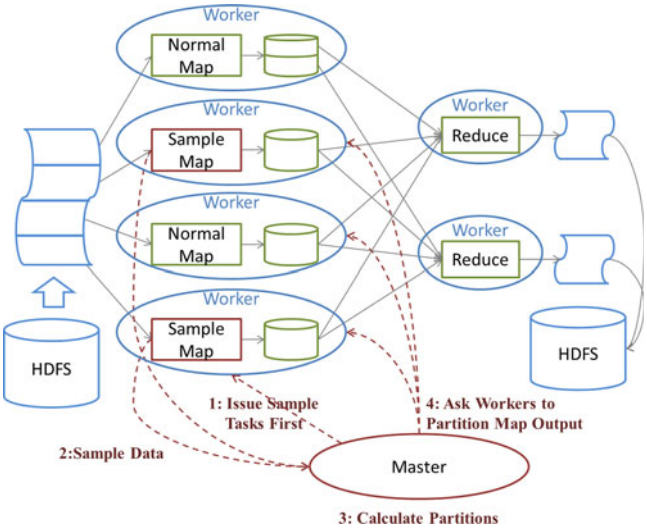


Fig. 2. System architecture.

(Page Rank, Inverted Index, etc.) and some simple applications (sort, grep, etc.). Mantri [22] has witnessed the data skew phenomenon in the Microsoft production cluster. They have found that the coefficients of variation in data size across tasks are 0.34 and 3.1 at the 50th and 90th percentiles, respectively. In the following, we show how LIBRA can address arbitrary reduce skew effectively.

3 The LIBRA System

In this section, we present a system which implements the LIBRA approach to solve data skew for general applications. The MapReduce framework we choose to implement LIBRA is Hadoop-1.0.0. The design goals of LIBRA include the following:

- *Transparency.* Data skew mitigation should be transparent to the users who do not need to know any sampling and partitioner details.
- *Parallelism.* It should preserve the parallelism of the original MapReduce framework as much as possible. This precludes any pre-run sampling of the input data and overlaps the map and the reduce stages as much as possible.
- *Accuracy.* its sampling method should be able to derive a reasonably accurate estimate of the input data distribution by sampling only a small fraction of the data.
- *Total order.* It should support total order of the output data. This saves applications which require such ordering an extra round of sorting at the end.
- *Large cluster splitting.* When application semantics permit, it should be able to split data associated with a single large cluster to multiple reducers while preserving the consistency of the output.
- *Heterogeneity consideration.* When the performance of the worker nodes is heterogeneous, it should be able to adjust the data partition accordingly so that all reducers finish around the same time.
- *Performance improvement.* Overall, it should result in significant improvement in application level performance such as the job execution time.

In the rest of this section, we will explain how LIBRA achieves the above goals.

3.1 System Overview

The architecture of our system is shown in Fig. 2. Data skew mitigation in LIBRA consists of the following steps:

- A small percentage of the original map tasks are selected as the *sample* tasks. They are issued first whenever the system has free slots. Other ordinary map tasks are issued only when there is no pending sample task to issue.
- Sample tasks collect statistics on the intermediate data during normal map processing and transmit a digest of that information to the master after they complete.
- The master collects all the sample information to derive an estimate of the data distribution, makes the partition decision and notifies the worker nodes.
- Upon receipt of the partition decision, the worker nodes need to partition the intermediate data generated by the sample tasks and already issued ordinary map tasks accordingly. Subsequently issued map tasks can partition the intermediate data directly without any extra overhead.
- Reduce tasks can be issued as soon as the partition decision is ready. They do not need to wait for all map tasks to finish.

3.2 Sampling and Partitioning

Since data skew is difficult to solve if the input distribution is unknown, a natural thought is to examine the data before deciding the partition. There are two common ways to do this. One approach is to launch some pre-run jobs which examine the data, collect the distribution statistics, and then decide an appropriate partition [2], [12], [23]. The drawback of this approach is that the real job cannot start until those pre-run jobs finish. The Hadoop range partitioner belongs to this category and as we will see in the experiments later, it can increase the job execution time significantly. The other approach is to integrate the sampling into the normal map process and generate the distribution statistics after all map tasks finish [13], [14], [15]. Since reduce tasks cannot start until the partition decision is made, this approach cannot take advantage of parallel processing between the map and the reduce phases.

We take a different approach by integrating sampling into a small percentage of the map tasks. We prioritize the execution of those sampling tasks over that of the normal map tasks: whenever the system has free slots, we launch the sampling tasks first. Since there are only a small percentage of them, they are likely to finish quite early in the map phase. There is an obvious trade-off between the sampling overhead and the accuracy of the result. In our experiments, we find that sampling 20 percent of the map tasks can generate a sufficiently accurate approximation for our purposes. Sampling beyond this threshold does not bring substantial additional benefit. Hence, we set the default sampling rate to 20 percent which can be changed by the user if necessary. To facilitate debugging, we want our execution to be reproducible across multiple runs of the same input data. Thus we

choose the fixed map tasks with the same step interval as sample tasks according to the sampling rate.

Within each sampling task, we also need to decide how much of the data it examines. Previous work can be divided into two categories on this: 1) examine the whole dataset processed by the task [13], [14], [15], [23], or 2) just sampling a small part of the input [2], [12]. The cost of the former category can be very high, but the result is more accurate. In contrast, the latter category can be much faster but provides a less accurate approximation. Our system belongs to the latter category. The next question is: what kind of sampling method to use? Commonly used sampling methods include the random, the interval and the split samplers provided by Hadoop [2] and TopCluster [15]. The random sampler is the most widely used, but it cannot achieve a good approximation to the distribution of real data in some cases, nor can TopCluster (shown in Fig. 8). Therefore, we develop a new sampling method which will be introduced in Section 4.

In case some sample map task happens to be stragglers or experience failure, we issue extra 10 percent more sample map tasks and consider the sample stage as finished when 90 percent of all sample tasks (i.e., sampling rate of all map tasks) complete. The master can then make a partition decision for this job and notify the decision ready event to the worker nodes. There are three major types of partitioners in previous work: hash, range and bin-packing [13], [14], [15]. The hash partitioner is the simplest but does not preserve total ordering and works well only with the uniform data distribution. The other two partitioners can both work well with most distributions, but only the range partitioner can provides a total ordered result. Hence, we use the range partitioner in our system.

In order not to delay the processing of the heartbeats from the worker nodes, we create a new thread to calculate the partition decision and save it to the distributed cache in Hadoop. The notification of the decision ready event contains only the ID of the job which is sent with the heartbeat responses of the worker nodes. By doing so, we can greatly reduce the overhead brought to the master node.

Once the partition decision has been computed by the master, the reduce tasks can be launched when free slots permit to take advantage of parallel processing between the map and the reduce phases.

3.3 Chunk Index for Partitioning

After the master notifies the worker nodes of the partition decision ready event, the worker nodes take responsibility for partitioning the intermediate data previously generated by the sampling tasks and already launched normal map tasks accordingly. This in general involves reading all the records from the intermediate output, finding the position of each partition key, and generating a small partition list which records the start and the end positions of each partition (shown in Fig. 3). When a reducer is launched later, the worker nodes can use the partition lists to help the reducer to locate and copy the data associated with its allocated key range from the map outputs quickly. The challenge here is how to find the partition breakpoints in a large amount of intermediate data. Since the intermediate data can be too large to fit into the memory, a brute force method using linear or binary search can be very time consuming: our experiments indicate that it can take up almost half of the map task execution time when the intermediate data is about the same size as the original input. Note that this is not a problem for map tasks issued after the partition decision is made: those tasks can generate the partition list as usual during their normal data processing (the same as the no skew mitigation case).

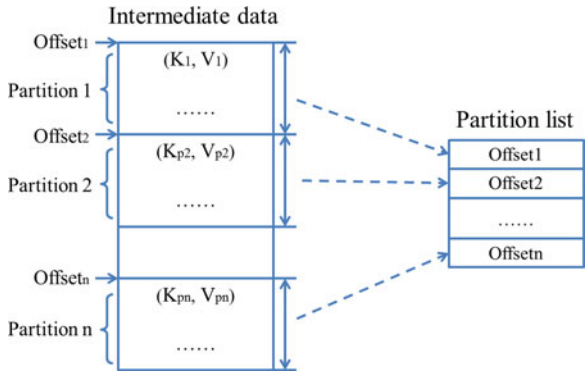


Fig. 3. Generate partition list.

To tackle this problem, we create a sparse index to speed up the search of the partition key positions when map tasks generate their intermediate data. We divide the intermediate data into multiple chunks (16 KB each in our current implementation) and generate a *sparse index* record for each chunk. The record includes the start key, the start position in the intermediate file, the raw length and the checksum of this chunk. The sparse index is small enough to fit into main memory and hence searching it can be performed efficiently. When we need to find the partition key positions in the intermediate data, we compare the partition key with the records in the sparse index first to find the data chunk containing it. Then we read the whole chunk into memory, examine the checksum, and find the accurate position of the key. By using this sparse index improvement, we can decrease the partition time by an order of magnitude. For example, we reduce the partition time from 3 to 5 seconds to 200 milliseconds when partitioning 64 MB intermediate data.

3.4 Splitting Large Cluster

The original MapReduce framework requires that a cluster (i.e., all tuples sharing the same key) be processed by a particular reducer. For applications that treat each intermediate key-value pair of a cluster independently in reduce phase, this can be overly restrictive. Some widely used examples are the sort and grep benchmark in the Hadoop distribution: the result would be the same even if a large cluster is split into multiple reducers for parallel processing. Another example is the *join* operation (broadcast join) commonly seen in database applications.

Enabling cluster splitting can have a profound impact on data skew mitigation. If cluster splitting is not allowed, an entire cluster have to be allocated as a whole to a single reducer. If some keys in the distribution are far more popular than others, it can be difficult for even the best skew mitigation algorithm to perform well. For example, suppose the intermediate data contain three keys: A, B and C. The count of them is 100, 10 and 10. Now we want to partition them into two reducers for

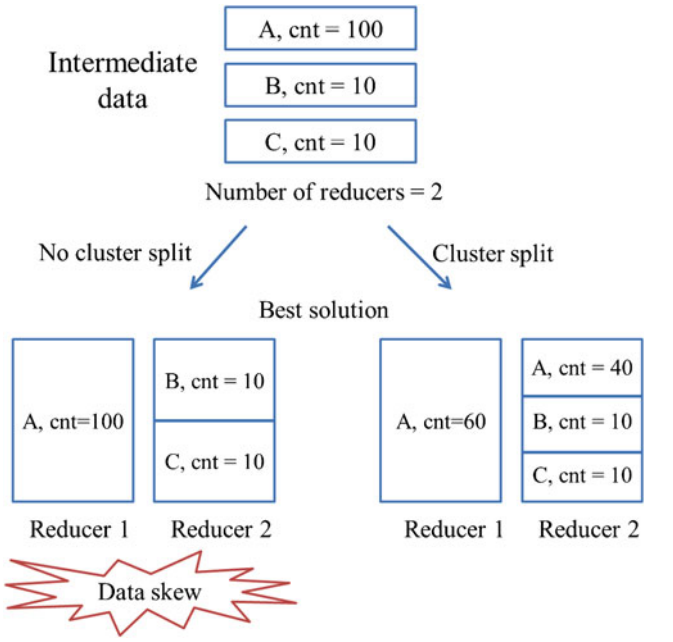


Fig. 4. Example of large cluster allocation.

processing. When cluster split is not allowed, the best solution is shown in left bottom corner of Fig. 4 which assigns the large key A to $reducer_1$ and the rest keys to $reducer_2$. As we can see, it still exhibits data skew.

Although [6] and [24] provide special methods to split large clusters in the join and CloudBurst applications (e.g., weighted range partitioning in [6]), they can only be used in specific applications and bring non-negligible extra sampling cost. To the best of our knowledge, none of the existing work provides a generic method for large cluster split when application semantics permit.

Based on this observation, we provide an effective cluster split strategy which allows large clusters to be split into multiple reduce tasks when appropriate. We modify the partition decision to include both the partition keys and the partition percentage. For example, a partition decision record (k, p) means that one of the partition point is p percent of key k. For map tasks issued before the partition decision is made, we can easily find these percentage partition points from the total-ordered intermediate outputs by adding some fields to the sparse index record. The new fields we add are the current record number in the key cluster K_{bi} and the total record count of K_{bi}. By calculating the ratio of current record number in K_{bi} to the total count of K_{bi}, we can get the key and the percentage in its cluster for the start record in each index block. In this way, we can quickly locate the index blocks which contain the partition points. For map tasks issued after the partition decision is made, we calculate a random secondary key in the range [0, 100] for each record and compare (key, secondary key) to partition decision records to decide which partition it belongs to (the order within the key may not be the same as input order). Using this cluster split strategy, the solution of the example shown in Fig. 4 can be optimized with 60 percent of the large key A to $reducer_1$ and the rest keys to $reducer_2$ (shown in right bottom corner). By doing so, the data skew is mitigated.

When application semantics permit, cluster splitting provides substantially more flexibility in mitigating the data skew. An application can indicate that it is amenable to cluster splitting by setting a parameterized flag when it begins execution.

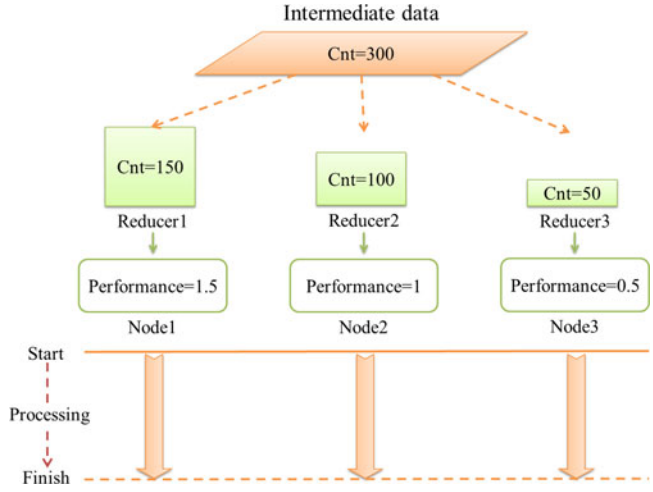


Fig. 5. Example of LIBRA partitioning in heterogeneous environment.

3.5 Heterogeneous Environment

The discussion so far has assumed that we should partition the data as evenly as possible. As observed in the introduction, not all reducers are created equal. Even if we assign them the same amount of data, their processing times can be different, depending on the performance of the worker nodes they run on. For example, a node could be "weaker" than others because it has a slower CPU or less computing resource at its disposal [25], or because it has a more complex dataset to work on. Microsoft has witnessed the variation of slow nodes over weeks due to the change of data popularity[22]. To fully exploit parallelism, we should equalize the amount of processing time for each reducer instead of equalizing the amount of data each processes. To the best of our knowledge, none of the existing data skew mitigation strategies has taken this into consideration.

LIBRA considers the performance of each worker node when partitioning the data. It assigns large tasks to fast nodes and small tasks to slow nodes so that all of them can be expected to finish around the same time. Fig. 5 gives an example on how LIBRA partitions its intermediate data in heterogeneous environments. This strategy can be useful even in the absence of data skew. Recall that speculative execution is a widely used approach to tackle the straggler problem in MapReduce: after it identifies a task as slow, it duplicates the task on another node where it hopefully can finish earlier. Speculative execution is reactive by its nature: it takes action after a task has already fallen behind other tasks. In contrast, we take a proactive approach to prevent stragglers from happening in the first place.

Like LIBRA, speculative execution requires a proper metric to measure the performance of a node since it needs to avoid duplicating tasks on slow nodes. Previously, LATE [26] uses the sum of progress of all the completed and running tasks on the worker node to represent the node performance, while Hadoop-0.21 uses the average progress rate of all the completed tasks on the node. However, these two metrics may cause some mistakes for some worker nodes

TABLE 1
The Average Coefficient of Variation

$p/qratio$	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50
$avg\ COV$	0.433	0.405	0.552	0.589	0.659	0.762	0.839	0.86	0.961	1.005

may do more time-consuming tasks and receive lower performance scores unfairly (the detailed analysis can be found in our previous work [27]). Therefore, the performance metric we choose for LIBRA is the moving average of the process bandwidth (the amount of data processed per second) of data-local map tasks (i.e., input data is located in a local worker) in the same job completed on the worker node. We have found it to be more stable and accurate in the MapReduce environment (a validation can be found in [27]). With the performance metric collected in each worker node, we adjust the range partition to assign nodes work based on their relative performance.

4 The LIBRA Algorithm

In this section, we present the sampling and partitioning algorithm in LIBRA. Our goal is to balance the load across reduce tasks. The algorithm consists of three steps:

1) Sample partial map tasks
2) Estimate intermediate data distribution
3) Apply range partition on the data

In the following, we will describe the details of these steps.

4.1 Problem Statement

We first give a formulation of our problem. The intermediate data between the map and the reduce phases can be represented as a set of tuples: $(K_1, C_1), (K_2, C_2), \ldots, (K_n, C_n)$, where K_i represents a distinct key in the map output, and C_i represents the number of tuples in the cluster of K_i. Without loss of generality, we assume that $K_i < K_{i+1}$ in the above list. Then our goal is to come up with a range partition on keys which minimizes the load of the largest reduce task. Let r be the number of reduce tasks. The range partition can be expressed as: $0 = pt_0 < pt_1 < \cdots < pt_r = n$ with reduce task i taking responsibility of keys in the range of $(K_{pt_{i-1}}, K_{pt_i}]$. Following the cost model proposed by previous work [12], [13], [14], we define the function $Cost(C_i)$ as the computational complexity of processing the cluster K_i in reduce tasks which must be specified by the users. For example, the cost function of the sort application can be estimated as $Cost(C_i) = C_i$ (for each cluster K_i, reducers only need to output C_i tuples directly). For reduce-side self-join application, the cost function should be C_i^2 since reducers need to output C_i tuples for each tuple in cluster K_i. By specifying the exact cost function, we can balance the execution time of each reducer one step further. Then the objective function can be expressed as follows:

$$\text{Minimize} \quad \max_{\mathrm{i}=1,2,\ldots,\mathrm{r}} \left\{ \sum_{\mathrm{j}=\mathrm{pt_{i-1}}+1}^{\mathrm{pt_i}} \mathrm{Cost(C_j)} \right\}. \tag{1}$$

Since the number of unique keys can be large, calculating the optimal solution to the above problem is unrealistic. Therefore, we present a distributed approximation algorithm by sampling and estimation.

4.2 Sampling Strategy

After a specific map task j is chosen for sampling, its normal execution will be plugged in with a lightweight sampling procedure. Along with the map execution, this procedure collects a statistic of (K_i^j, C_i^j) for each key K_i^j in the output of this task, where C_i^j is the frequency (i.e., the number of records) of key K_i^j. Since the number of such (K_i^j, C_i^j) tuples can be on the same order of magnitude as the input data size, we keep only a sample set S_{sample} containing the following two parts:

- $S_{largest}$: p tuples with the largest C_i^j.
- S_{normal}: q tuples randomly selected from the rest according to uniform distribution (excluding tuples in $S_{largest}$).

This sampling task then transmits the following statistics to the master: the sample set $S_{sample} = S_{largest} \cup S_{normal}$, the total number of records (TR^j) and the total number of distinct clusters (TC^j) generated by this task. The size of the sample set $p + q$ is constrained by the amount of memory and the network bandwidth at the master. The larger $p + q$ is, the more accurate approximation to the real data distribution we will achieve. In practice, we find that a small $p + q$ value (e.g., 1,000) has already reached a good approximation and brings negligible overhead (shown in the Section 5).

The ratio of p/q is positively related to the degree of the data skew: the heavier the skew, the larger the ratio should be. To select a good ratio, we generate 10 GB synthetic datasets following Zipf distributions with varying σ parameters (from 0.2 to 1.4) to control the degree of the skew. Larger σ value means heavier skew. We run the sort benchmark and set the number of reduce tasks to 30. Fig. 7 shows the coefficient of variation in data size across reduce tasks with different p/q ratio and skew degree σ. From the result, we can find that when σ is low (e.g., 0.2), the optimal p/q ratio is low, meaning that sampling more random keys would be better. However, when σ is high (e.g., 1.4), the optimal p/q ratio is also high, meaning that sampling more large keys can divide the intermediate data more evenly. In order to find a good p/q ratio which works reasonably well with a wide variety of the datasets, we calculate the average coefficient of variation ($avg\ COV$) of all σ ($\sigma \in S$) for each p/q ratio as follows:

$$avg\ COV_{p/q} = \frac{\sum_{\sigma \in S} COV_{\sigma}^{p/q}}{|S|}. \tag{2}$$

The result is shown in Table 1. From the result, we can find that the optimal p/q ratio which can work well in both skew and non-skew cases is 0.10. Therefore, we set the default p/q ratio to 0.10.

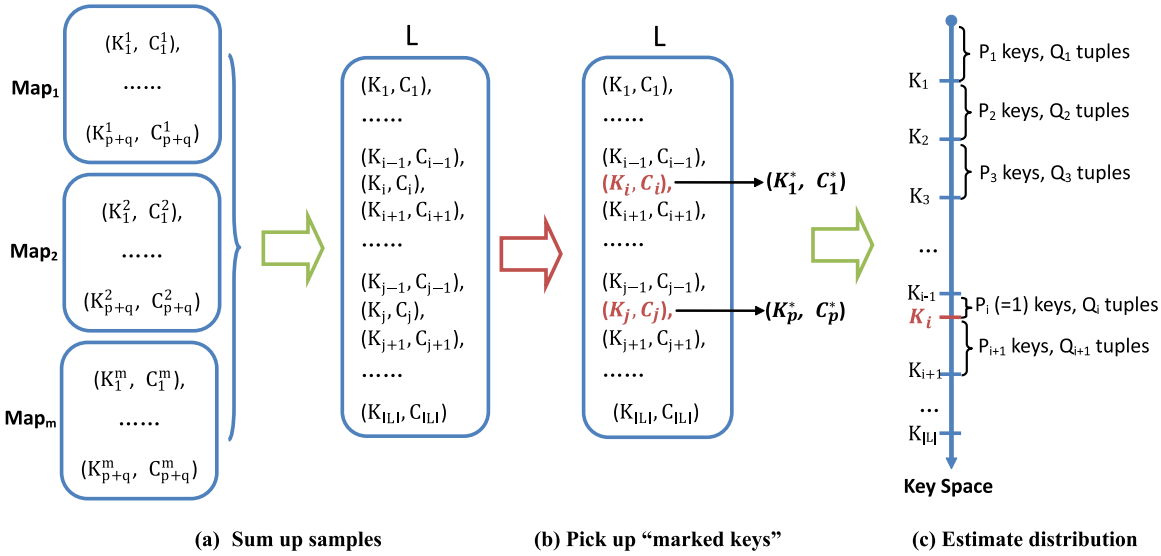


Fig. 6. LIBRA sampling and distribution estimation.

4.3 Estimate Intermediate Data Distribution

After the completion of all sample map tasks, the master aggregates the sampling information in the above step to estimate the distribution of the data. The main steps of the estimation can be shown in Fig. 6. It first combines all the sample tuples with the same key into one tuple (K_i, C_i) by adding up their frequency (shown in Fig. 6a). It then sorts these combined tuples to generate an aggregated list L. Suppose there are m maps for sampling and $S^j_{sample} = \{(K^j_l, C^j_l)\}, l = 1, 2, \ldots, p+q$ is the sample set of map j. Then the aggregated list L is:

$$L = \left\{ \left(K_i, C_i = \sum_{j=1}^{m} \{C^j_l | K^j_l = K_i\} \right) \right\}, \quad K_i < K_{i+1}. \tag{3}$$

To calculate the total number of records TR, we simply sum up the record counts in all sample map tasks. However, calculating the total number of distinct clusters TC is hard because clusters processed by different map tasks may share the same key and hence should not be counted twice.

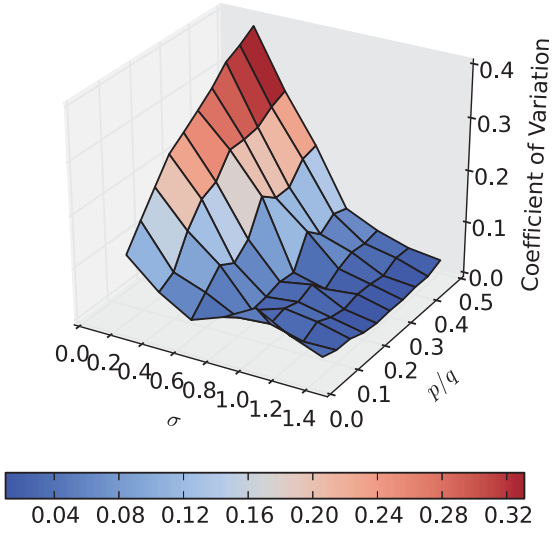


Fig. 7. Coefficient of variation with different p/q ratio and skew degree σ.

For example, assume that there are two sample map tasks and their sample sets are: $\{(A,10), (B,5), (C,3), (D,2), (E,2)\}$, $\{(A,20), (B,3), (D,1), (F,1), (H,1)\}$, in which $p = 2$ and $q = 3$. By summing up the frequencies of the same key, the merged sample set S_{sample} is $\{(A,30), (B,8), (C,3), (D,3), (E,2), (F,1), (H,1)\}$. Suppose that there are 50 keys and 10,000 records in total in the first sample map task, while there are 60 keys and 15,000 records in the second sample map task. Apparently, aggregated TR of these two sample tasks equals to 25,000. However, aggregated TC of them is difficult to calculate because some keys may exist in both map tasks (such as key A, B sand D).

To address this, we estimate the overlap degree of each sample set S^j_{sample} (the sample set of map j) with the overall distribution L (only the normal part) and weight the contribution to TC by this amount. The calculation of TR and TC can be expressed as follows:

$$TR = TR^\star + TR^j, \tag{4}$$

$$TC = (TC^\star + TC^j - 2p) * \left(1 - \frac{Degree}{2}\right) + p, \tag{5}$$

$$Degree = \frac{2 * (|L^\star| + |S^j_{sample}| - |L| - p)}{|L^\star| + |S^j_{sample}| - 2p}, \tag{6}$$

where $TR^\star$, $TC^\star$, and $L^\star$ represent the aggregated result before incorporating the sample information of map j, while TR, TC, and L represent the result after aggregating map j. TR^j and TC^j represent the number of total records and the number of total exclusive clusters of map j. $Degree$ represents the overlap degree of map j and the aggregated result. The larger $Degree$ is, the more consistent they are. In the example above, we calculate the aggregated TC of two sample tasks as follows: Since each sample set has three normal keys, among which only one key is shared (key D), $Degree$ can be calculated as $\frac{2*1}{3+3} = \frac{1}{3}$. As a result, the estimated

distinct keys in these two map tasks can be calculated as $TC = (50 + 60 - 2 * 2) * (1 - \frac{1}{3*2}) + 2 = 90$.

Next we estimate the distribution of the intermediate data. Let P_i be the approximate number of keys in the range of $(K_{i-1}, K_i]$ and Q_i be the approximate frequency of each key in this range. We estimate the distribution (P_i, Q_i) according to L as follows:

i) Pick up p keys from L with the largest C_i as the "marked keys", denoted as $(K_1^\star, C_1^\star), \ldots, (K_p^\star, C_p^\star)$. In the example above, the "marked keys" are $(A, 30)$ and $(B, 8)$. This procedure can be demonstrated in Fig. 6b. Since all the locally largest p clusters have been picked up in the sample map tasks, for each marked key $K_i^\star$, $P_i^\star$ is set to 1, and $Q_i^\star$ is approximated by $C_i^\star$ in the aggregated list.

ii) Suppose $TC^L = |L|$, $TR^L = \sum_{(K_i, C_i) \in L} C_i$. For the other $TC_{normal} = TC - p$ keys and $TR_{normal} = TR - \sum_{i=1}^{p} C_i^\star$ records, we estimate their frequencies as follows:
- Since normal clusters are randomly selected, we proportionally spread all the rest $TC^L_{normal} = TC^L - p$ keys and $TR^L_{normal} = TR^L - \sum_{i=1}^{p} C_i^\star$ records in L over the ranges partitioned by marked keys.
- Then for each normal key K_i in L, we have: $P_i = \frac{TC_{normal}}{TC^L_{normal}}$, $Q_i = \frac{C_i \times TR_{normal}}{P_i \times TR^L_{normal}}$.

This step can be shown in Fig. 6 c.

4.4 Range Partition

We adopt the above approximation to the data distribution to get an approximate solution to the range partition. We need to generate a list of partition points in the aggregated list L where $0 = pt_0 < pt_1 < \cdots < pt_r = |L|$ and minimize:

$$\max_{i=1\ldots r} \left\{ \sum_{j=pt_{i-1}+1}^{pt_i} \{P_j \times Cost(Q_j)\} \right\}. \quad (7)$$

We use dynamic programming to solve this optimization problem: let $F(i, j)$ represent the minimum value of the largest partition sum of cutting the first i items into j partitions, and $W(a, b) = \sum_{l=a}^{b} \{P_l \times Cost(Q_l)\}$. Then the recursive formulation of $F(i, j)$ is:

$$F(i, j) = \min_{k=j-1\ldots i-1} \{\max\{F(k, j-1), W(k+1, i)\}\}. \quad (8)$$

The partition decision can be derived from optimized decision of $F(i, j)$. The time complexity of calculating the above equation is $O(n^2 r)$, where n is the length of the aggregated list and r is the number of reducers. We optimize the brute force calculation by the following two theorems:

Theorem 1. *For specific i and j, define $f_i(k) = \max\{F(k, j-1), W(k+1, i)\}, k = j-1, \ldots, i-1$. Then $f_i(k)$ is an unimodal function with the minimal point.*

Proof. With parameter k, $F(k, j-1)$ is a monotonically increasing function and W(k+1, i) is a monotonically decreasing function by definition. So it is obvious that the compound function by maximizing the value of these two functions is an unimodal function. The minimal point will appear either at the intersection of these two functions or at the endpoints of the defining range of parameter k. □

Theorem 2. *For specific j, define $d(i) = k_{min}$ s.t. $f_i(k_{min})$ is the minimal point (if there are multiple k to get minimal points, k_{min} is the smallest one). Then we have $d(i) \geq d(i-1)$ for each $i = j+1, \ldots, n$.*

Proof. Suppose we have $d(i) < d(i-1)$. According to the definition of $d(i-1)$, we have:

$$\begin{aligned}
&f_{i-1}(d(i-1)) < f_{i-1}(d(i)) \\
&\quad => \max\{F(d(i-1), j-1), W(d(i-1)+1, i-1)\} \\
&\qquad < \max\{F(d(i), j-1), W(d(i)+1, i-1)\} \\
&\quad => \max\{F(d(i-1), j-1), W(d(i-1)+1, i-1) + P_i \\
&\qquad \times Cost(Q_i)\} < \max\{F(d(i), j-1), W(d(i)+1, i-1) \\
&\qquad + P_i \times Cost(Q_i)\} \\
&\quad => \max\{F(d(i-1), j-1), W(d(i-1)+1, i)\} \\
&\qquad < \max\{F(d(i), j-1), W(d(i)+1, i)\} \\
&\quad => f_i(d(i-1)) < f_i(d(i)),
\end{aligned}$$

which is contradictory to the definition of $d(i)$. Hence we have $d(i) \geq d(i-1)$. □

According to the two theorems, when we calculate $F(i, j)$ by increasing parameter i, the optimized decision $d(i)$ is also increased. Using this property we can improve this algorithm to $O(nr)$, which makes it highly efficient in practice.

Heterogeneous environments are more complicated. We model it as follows: we define e_i as the performance factor of the ith worker node ($e_i = \frac{\text{Performance}_{\text{node}_i}}{\text{AvgPerformance}}$). And we restrict the ith partition to be handled on the ith worker node. Then the object is modified to minimize:

$$\max_{i=1\ldots r} \left\{ \frac{\sum_{j=pt_{i-1}+1}^{pt_i} \{P_j \times Cost(Q_j)\}}{e_i} \right\}. \quad (9)$$

Then we can use the same algorithm to obtain the optimized range partition in heterogeneous environments.

5 Evaluation

In this section, we evaluate the performance of LIBRA on some popular applications with both synthetic and real-world datasets under both homogeneous and heterogeneous environments. We find that:

i) LIBRA sampling method achieves a good approximation to the distribution of the original whole dataset (Section 5.2).

ii) LIBRA can partition the intermediate data more evenly across reduce tasks and reduce the variability of job execution time significantly (Section 5.4).

iii) LIBRA can be widely used in various applications and deliver up to a factor of 4X performance improvement (Section 5.5).

iv) LIBRA can fit well in both homogeneous and heterogeneous environments (Section 5.6).

v) The overhead of LIBRA is minimal.

5.1 Experiment Environment

We set up our Hadoop cluster with 15 servers. Each server contains dual-Processors (2.4 GHz Xeon E5620), 24 GB of RAM, and two 150 GB disks. They are connected by 1 Gbps Ethernet and managed by the OpenStack Cloud Operating System [28]. We use the KVM virtualization software [29] to construct medium sized VMs with two virtual core, 4 GB RAM and 30 GB of disk space. We conduct our experiments in a homogeneous environment with two VMs running on each server to avoid heavy resource competition. Later in the section, we will change this environment into a heterogeneous one by running a set of CPU and I/O intensive processes on a subset of the physical machines to emulate resource competition. All experiments use the default configuration in Hadoop for HDFS and MapReduce except otherwise noted (e.g., the HDFS block size is 64 MB, max Java heap size is 2 GB, and sort buffer size is 100 MB). Therefore, there are 30 worker nodes in our Hadoop cluster which contain an aggregate of 60 map slots and 60 reduce slots. We evaluate the following applications.

Sort. We use the sort benchmark in Hadoop as our main workload because it is widely used and represents many kinds of data-intensive jobs. We generate 10 GB synthetic datasets following Zipf distributions with varying σ parameters to control the degree of the skew. We choose Zipf distribution workload because it is very common in the data coming from the real world, e.g., the word occurrences in natural language, city sizes, many features of the Internet [30], the sizes of craters on the moon [19].

Grep. Grep is a popular application for large scale data processing. It searches some regular expressions through input text files and outputs the lines which contain the matched expressions. We modify the grep benchmark in Hadoop so that it outputs the matched lines in a descending order based on how frequently the searched expression occurs. The dataset we used is the full English Wikipedia archive with the total data size of 31GB.

Inverted Index. Inverted indices are widely used in search area. We implement a job in Hadoop that builds an inverted index from given documents and generates a compressed bit vector posting list for each word. We use the Potter word stemming algorithm and a stopword list to pre-process the text during the map phase, and then use the RADIX partitioner to map alphabet to reduce tasks in order to produce a lexicographically ordered result. The dataset we used is also the full English Wikipedia archive.

Join. Join is one of the most common applications that experience the data skew problem. We implement a simple broadcast join job in Hadoop which partitions a large table in the map phase, while a small table is directly read in the reduce phase to generate a hash table for speeding up join operation. When the small table is too large to fit into the memory, we use a buffer to keep only a part of the small table in memory and use the cache replacement strategy to update the buffer. We use synthetic datasets which follow Zipf distribution to generate the large tables, while use datasets which follow either the uniform distribution or the Zipf distribution to generate the small tables.

We run each test case at least three times and take the average value in order to reduce the influence of the variable environment. We compare LIBRA with Hadoop hash partition, Hadoop range partition, some application specific partition methods and SkewTune [16]. We compute the coefficient of variation in data size across reduce tasks to measure the effectiveness of skew mitigation. For SkewTune, we compute the coefficient of variation in data size processed by different worker nodes. The smaller the coefficient, the better.

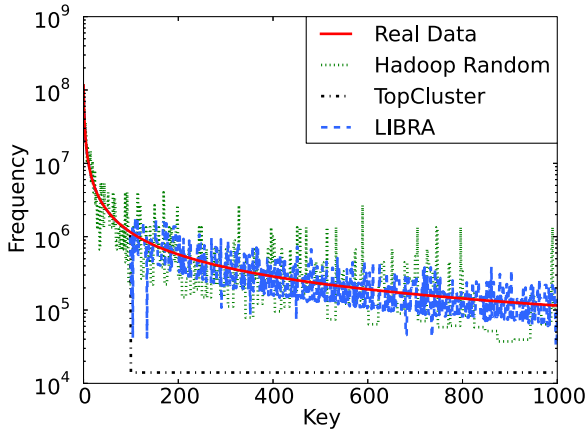


Fig. 8. Comparison of three sampling methods in sort.

5.2 Accuracy of the Sampling Method

To evaluate how our sampling method can achieve a good approximation to the original data distribution, we run a sort benchmark with 10 GB synthetic dataset which follows Zipf distributions ($\sigma = 1.0$). We set the number of reduce tasks to 30, and compare our sampling method with the Hadoop random sampler and the TopCluster sampling method [15]. All three methods sample 20 percent of input splits and 1,000 keys from each split. The master keeps the same number of large clusters for LIBRA and TopCluster.

To give a rough idea of the accuracy of the sampling methods, we calculate the root mean square (rms) error

$$\sqrt{\frac{\sum_{i=1}^{n}(x_i^{appro}-x_i^{real})^2}{n}}$$

of each sampling method for all 65,535 keys in the original data. The rms errors for LIBRA, TopCluster, and Hadoop random sampler are 183,278, 333,953, and 917,065, respectively. This demonstrates that our sampling method is far more accurate than the other two. This is visualized in Fig. 8 for the top 1,000 large keys in the data (y-axis in log scale). Note that the TopCluster curve has a flat, long tail. In this method, each map task samples the large clusters in its processed data. The master aggregates information of large keys from all map tasks and assumes that small keys are uniformly distributed. The curve shows that it has a fairly accurate estimate on the large keys (the beginning part of the curve), but its assumption of the uniform distribution can be misleading when there are a large number of small keys in the data (the rest of the curve). Its lack of information on the small cluster makes it difficult to generate accurate range partition if the optimal partition breakpoints happen to be on small clusters. The figure shows that LIBRA can achieve a better approximation to the original data distribution.

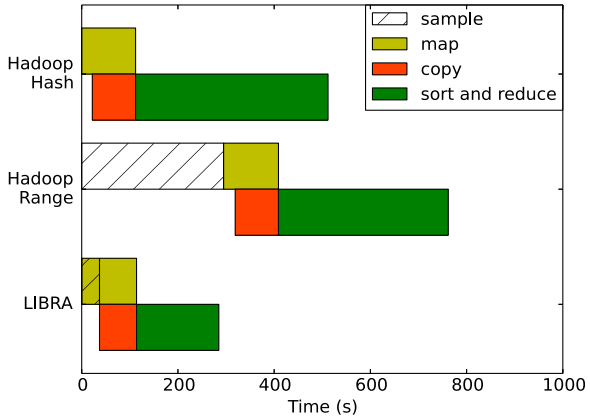


Fig. 9. Comparison of job execution time in sort.

5.3 Job Execution Time

A major motivation for data skew mitigation is to improve the job execution time. This is shown in Fig. 9 where we compare the execution time of our system with that of the two strategies in Hadoop. As we can see from the figure, the improvement is dramatic: the execution speed in our system is 80 percent faster than that in Hadoop hash. This comparison is actually unfair to us because Hadoop hash (unlike ours) cannot support total order of the output data. When compared to Hadoop range (which supports the total order), our improvement jumps to 167 percent. In fact, by the time Hadoop finishes its pre-run sample of the input data, our system has already completed its entire execution.

The figure also shows that the overhead of our sampling method is negligible: our combined sample/map phase is about the same length as the map phase in Hadoop hash and Hadoop range (which perform no sample). This demonstrates the efficiency of our carefully designed algorithm. Our much improved execution time in the reduce phase is because LIBRA can partition the intermediate data much more evenly (as evidenced in Fig. 1 at the beginning of the paper). The coefficient of variation among all reduce tasks in LIBRA is only 0.07, while in Hadoop range and Hadoop hash it reaches 0.47 and 0.51, respectively.

5.4 Degrees of the Data Skew

To see how our system performs when the input data exhibits different degrees of skew, we repeat the previous experiment but with σ varying from 0.2 to 1.2. Fig. 10 shows how the job execution time and the coefficient of variation change when the skew increases. For the two strategies in Hadoop (marked 'Hadoop_hash' and 'Hadoop_range') and SkewTune (We use Hadoop hash as the original partitioner), both metrics increase substantially once the degree of the skew reaches a certain threshold. We compare them with two versions of LIBRA in this experiment.

Recall that one optimization in LIBRA is to split a large cluster across multiple reducers. The figure shows the performance of our system with and without this optimization (marked 'LIBRA_CSP' and 'LIBRA_NCSP', respectively). As we can see from the figure, this optimization has a profound impact on partitioning the data evenly. With this optimization, both the job execution time and the coefficient of variation remain very low as σ increases. (We have continued the experiments for σ up to 2.0 and find that the curves remain flat.) Without this optimization, the curves begin to climb up after σ reaches a certain threshold. Even so, our system performs better than Hadoop hash and much better than Hadoop range and SkewTune. The reason that SkewTune performs worse than Hadoop hash is that SkewTune does not detect or split large keys and hence cannot make a better partition decision. Moreover, it brings extra overhead (e.g., resource competition). As explained earlier, Hadoop hash does not support total order of the output data, while LIBRA does. Hence, we consider the result quite remarkable even without the cluster split optimization. From this experiment, we can see that the overhead of LIBRA is negligible even in the absence of skew ($\sigma = 0.2$).

We also run this experiment with a large scale synthetic dataset of 100 GB on a large scale homogeneous cluster consisting of 100 medium sized VMs running across 20 servers. Fig. 11 shows how the job execution time and the coefficient of variation change in Hadoop (marked 'Hadoop_hash' and 'Hadoop_range'), SkewTune and LIBRA (marked 'LIBRA_CSP' and 'LIBRA_NCSP'). As we can see, both metrics increase rapidly when the degree of the skew exceeds 0.6 for Hadoop_hash, Hadoop_range, SkewTune, and LIBRA_NCSP. In contrast, with LIBRA_CSP, both of the job execution time and the coefficient of variation stay at a very low level. Even without the cluster split optimization, our LIBRA_NCSP performs better than Hadoop_hash, Hadoop_range, and SkewTune due to its more reasonable partition strategy. This experiment also demonstrates the negligible overhead of LIBRA when the scale of the dataset is large.

5.5 Grep, Inverted Index, and Join

Next we evaluate our system with the grep, the inverted index, and the join applications described at the beginning

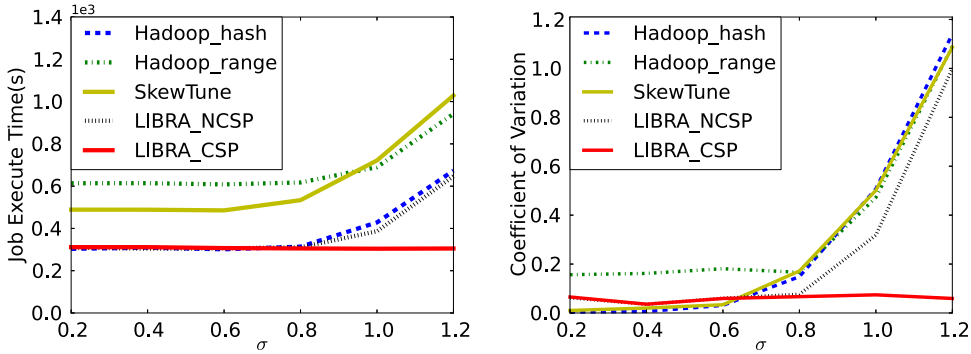


Fig. 10. Job execution time (left) and coefficient of variation (right) as the degree of data skew increases in sort.

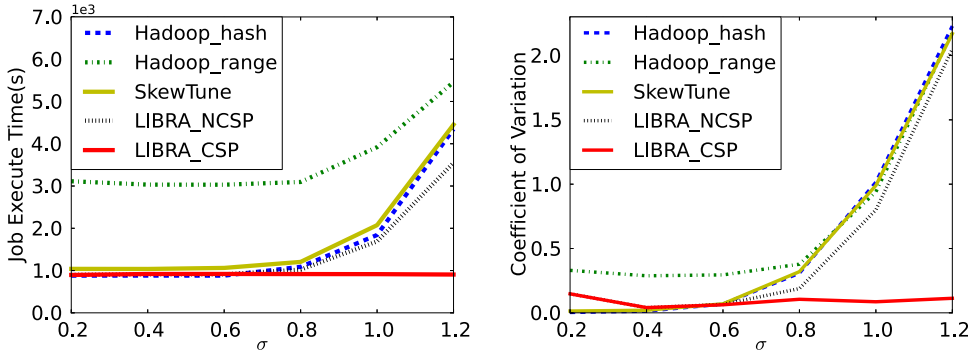


Fig. 11. Job execution time (left) and coefficient of variation (right) as the degree of skew increases in large scale sort.

of this section. First, we run the grep benchmark with the full English Wikipedia archive dataset. Since the behavior of grep depends on how frequently the search expression appears in the input file, we tune the search expression so that its output percentage varies from 10 to 100 percent of the input. Fig. 12 shows how the job execution time and the coefficient of variation change when the output percentage increases.

Note that Hadoop does not provide a suitable range partitioner for this application: its pre-run sampler samples the input data and cannot handle applications where the intermediate data is of a different format from the input. In contrast, LIBRA samples the intermediate data directly and works well for all types of applications. The figure compares the performance of LIBRA with and without the cluster splitting optimization with Hadoop hash.

As we can see from the figure, LIBRA with cluster split enabled performs significantly better than Hadoop hash when the output percentage of grep is low. This is because searching unpopular words in the archive tends to generate results with heavy data skew where LIBRA has a clear advantage over Hadoop. When the output percentage is high, the resulting data become more evenly distributed and hence the performance difference becomes smaller, although LIBRA is still slightly better. When the cluster split optimization is disabled, the performance of LIBRA becomes similar to, but still slightly better than Hadoop hash. Again, this is already impressive given that LIBRA supports total order while Hadoop hash does not.

For the inverted index test, we also use the full English Wikipedia archive with a total data size of 31 GB. As a target for comparison, we use the RADIX partition method to generate a lexicographically ordered result which sets the number of reducers to 27: one for special characters and the other 26 for words starting with each letter of the alphabet. We also compare with SkewTune which uses RADIX as the original partitioner. Fig. 13 compares the reduce time (from the time the last map task finishes to the time the last reduce task finishes) of the RADIX partition, SkewTune and LIBRA with or without cluster split enabled. The results show that LIBRA can partition the intermediate data much more evenly (almost a factor of 4 improvement over RADIX and a factor of 2 improvement over SkewTune) and that the cluster split optimization has little impact on this application.

To run the join application, we set up the datasets for large tables using Zipf distribution ($\sigma = 1.0$), while using the uniform or the Zipf distribution for small tables. We set up three test cases: a) a large table joins a large table (200M $*$ 200M): one of them follows the Zipf distribution and the other follows the uniform distribution, b) a large table joins a small table (2G $*$ 2M): the large one follows the Zipf distribution and the small one follows the uniform distribution, c) a small table joins a small table (20M $*$ 2M): both tables follow the Zipf distribution.

We compare LIBRA (using broadcast join described in Section 5.1) with Hash Join (PHJ), Skewed Join (PSJ) and Replicated Join (PRJ) in Pig [31]. Fig. 14 shows the job execution time of these three test cases. In case (a), the best scheme in Pig is PSJ, which samples a large table to generate the key distribution and makes the partition decision beforehand. The left figure shows that LIBRA can perform almost three time faster than PSJ. In case (b), the best join scheme in Pig is PRJ, which splits the large table into multiple map tasks and performs the join in map tasks by reading the small table directly into memory. The middle figure shows that LIBRA outperforms PRJ due to better

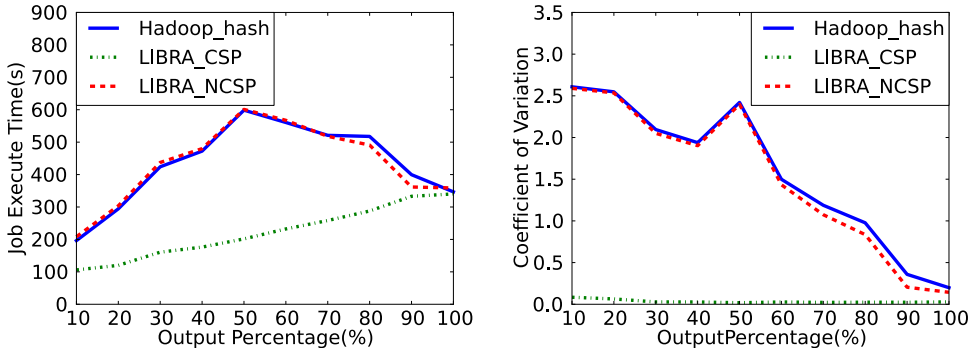


Fig. 12. Job execution time (left) and coefficient of variation (right) as the output percentage increases in grep.

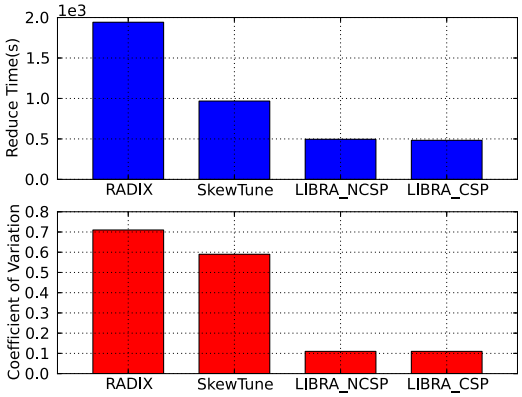


Fig. 13. Reduce phase of inverted index application.

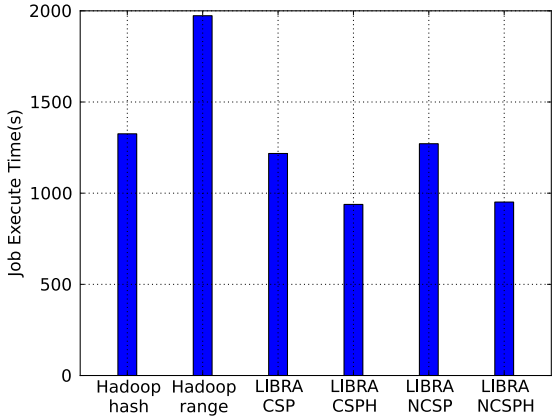


Fig. 15. Job execution time of sort in heterogeneous environments.

parallelism by setting more reducers while the parallelism in replicated join is limited by the number of map tasks allocated by the MapReduce system. In case (c), we compare LIBRA with all three join schemes in Pig. The right figure shows that LIBRA is 2.8 times faster than PHJ (the default scheme in Pig) and PSJ, and is five times faster than PRJ.

5.6 Heterogeneous Environments

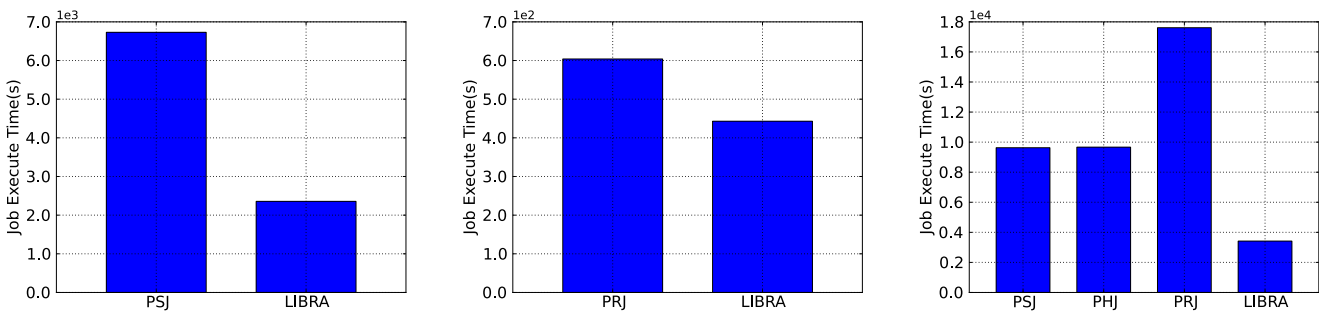
To show LIBRA can fit well with the variable cloud computing environments, we set up a heterogeneous test environment by running a set of CPU and I/O intensive processes (e.g., heavy scientific computation and *dd* process which creates large files in a loop to write random data) to generate background load on two of the servers. We use sort benchmark with ($\sigma = 0.2$). We intentionally choose a small σ value so that all methods can partition the intermediate data quite evenly. This allows us to focus on the impact of environment heterogeneity. Fig. 15 shows the results for four versions of LIBRA: with or without considering environment heterogeneity, and with or without cluster split enabled. As we can see from the figure, LIBRA with heterogeneity consideration (LIBRA CSPH and NCSPH) can perform 30 and 34 percent faster than LIBRA without this consideration (LIBRA CSP and NCSP). It also shows that (not surprisingly) enabling cluster split (LIBRA CSP and CSPH) has little impact for this workload. The default configuration of LIBRA (CSPH) can perform 41 percent faster than Hadoop hash and 110 percent faster than Hadoop range.

6 Related Work

The data skew problem is a common and important problem that needs to be solved in distributed systems. It has been studied in the parallel database area, but only limited on join [5], [6], [7], [8], [9], group [10], and aggregate [11] operations. Some of these technologies have already been carried to MapReduce, like SkewedJoin in Pig [32]. However, users generally still need to implement their own methods for their specific applications to tackle the data skew, such as CloudBurst [24] and SkewReduce [12].

Data skew has also been studied in the MapReduce environment during the past three years. Okcan et al. propose a skew optimization for the theta join by adding two pre-run sampling and counting jobs. Kwon et al. provide a system called SkewReduce which optimize the data partition for the spatial feature extraction application by operating preprocessing extracting and sampling procedures [12]. Although these solutions can mitigate data skew to some extent, they have significant overhead due to the pre-run jobs and are applicable only to certain applications.

Researchers have also tried to collect data information during the job execution. Ibrahim et al. have studied the locality-aware and fairness-aware key partition optimization for reduce by collecting key frequency information in each node and aggregating them on the master after all maps done [13]. They sort all keys by their $\frac{Fairness}{Locality}$ value and greedily choose the reduce node with the maximum fairness score for each key. Gufler et al. partition the intermediate data into more partitions than the number of reducers, and then use a greedy bin-packing method to allocate them to the set of reducers after all map tasks finish [14]. Later, they propose a sampling method called TopCluster to approximate the distribution of the input data [15]. In TopCluster, each map task samples the largest clusters represented by



(a) 200M (zipf) * 200M (uniform) (b) 2G (zipf) * 2M (uniform) (c) 20M (zipf) * 2M (zipf)

Fig. 14. Performance of join applications.

keys and their frequencies, and delivers them to the master for aggregation. As a result, the master can calculate the partition cost more accurately according to the global information of large keys with the assumption that small keys are uniform distributed. However, none of the above approaches can start shuffling for reduce tasks until all maps complete. Therefore, they cannot take advantage of parallel processing between the map and the reduce phases. Moreover, they use the bin-packing partitioner which provides poor support for the total ordered applications.

The SkewTune system tackles the data skew problem from a different angle [16]. It does not aim to partition the intermediate data evenly at the beginning. Instead, it adjusts the data partition dynamically: after detecting a straggler task, it repartitions the unprocessed data of the task and assigns them to new tasks in other nodes. It reconstructs the output by concatenating the results from those tasks according to the input order. SkewTune and LIBRA are complementary to each other. When load change dynamically or when reduce failure occurs, it is better to mitigate skew lazily using SkewTune. On the other hand, when the load is relatively stable, LIBRA can better balance the copy and the sort phases in reduce tasks and its large cluster split optimization can improve the performance further when application semantics permit.

Previous work also exists on tackling the straggler problem in MapReduce. The straggler problem was first studied by Dean et al. in [1]. They use speculative execution to back up the last few running tasks and have observed that it can decrease the job execution time by 44 percent. The original speculative execution strategy in Hadoop identifies a task as a straggler when the task's progress falls behind the average progress of all tasks by a fixed gap. Zaharia et al. have found that this does not fit well in heterogeneous environments and proposed a new strategy called LATE [26], which calculates the progress rate of tasks and selects the slow task with the longest remaining time to back up. Later, Ganeshi et al. propose a new method called Mantri [22] which uses the task's *process bandwidth* to calculate the task's *remaining time*. It also considers saving cluster computing resource in its strategy. However, our earlier work finds that there still exist several scenarios that will affect the performance of the above strategies. Therefore, we develop a new strategy called MCP [27] which divides a task into multiple phases and uses both the predicted progress rate and process bandwidth within a phase to identify slow tasks more accurately and promptly. In addition, MCP takes the load of the cluster into consideration and uses a cost-benefit model to determine which task is worth backing up. When choosing the backup destination, MCP also pays attention to data locality and data skew. All approaches above can only solve stragglers due to environment heterogeneity. Unlike LIBRA, they cannot solve the data skew problem.

7 Conclusions

Data skew mitigation is important in improving MapReduce performance. This paper has presented LIBRA, a system that implements a set of innovative skew mitigation strategies in an existing MapReduce system. One unique feature of LIBRA is its support of large cluster split and its adjustment for heterogeneous environments. In some sense, we can handle not only the data skew, but also the reducer skew (i.e., variation in the performance of reducer nodes). Performance evaluation in both synthetic and real workloads demonstrates that the resulting performance improvement is significant and that the overhead is minimal and negligible even in the absence of skew.

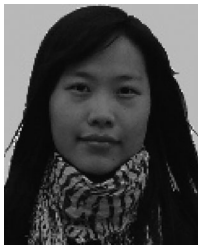
Acknowledgments

The authors would like to thank the anonymous reviewers for their invaluable feedback. This work was supported by the National High Technology Research and Development Program ("863" Program) of China (Grant No.2013AA013203) and the National Natural Science Foundation of China (Grant No. 61170056). The contact author is Zhen Xiao.

References

[1] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.
[2] Apache hadoop [Online]. Available: http://lucene.apache.org/hadoop/, 2013.
[3] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed data-parallel programs from sequential building blocks," in *Proc. ACM SIGOPS/EuroSys Eur. Conf. Comput. Syst.*, 2007, pp. 59–72.
[4] Y. Kwon, M. Balazinska, and B. Howe, "A study of skew in mapreduce applications," in *Proc. Open Cirrus Summit*, 2011.
[5] C. B. Walton, A. G. Dale, and R. M. Jenevein, "A taxonomy and performance model of data skew effects in parallel joins," in *Proc. Int. Conf. Very Large Data Bases*, 1991, pp. 537–548.
[6] D. J. DeWitt, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Practical skew handling in parallel joins," in *Proc. Int. Conf. Very Large DataBases*, 1992, pp. 27–40.
[7] J. W. Stamos and H. C. Young, "A symmetric fragment and replicate algorithm for distributed joins," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, no. 12, pp. 1345–1354, 1993.
[8] V. Poosala and Y. E. Ioannidis, "Estimation of query-result distribution and its application in parallel-join load balancing," in *Proc. Int. Conf. Very Large Data Bases*, 1996, pp. 448–459.
[9] Y. Xu and P. Kostamaa, "Efficient outer join data skew handling in parallel dbms," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1390–1396, 2009.
[10] S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2000, pp. 487–498.
[11] A. Shatdal and J. F. Naughton, "Adaptive parallel aggregation algorithms," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 1995, pp. 104–114.
[12] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skew-resistant parallel processing of feature-extracting scientific user-defined functions," in *Proc. ACM Symp. Cloud Comput.*, 2010, pp. 75–86.
[13] S. Ibrahim, J. Hai, L. Lu, W. Song, H. Bingsheng, and Q. Li, "Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud," in *Proc. IEEE Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 17–24.
[14] G. Benjamin, A. Nikolaus, R. Angelika, and K. Alfons, "Handling data skew in mapreduce," in *Proc. Int. Conf. Cloud Comput. Serv. Sci.*, 2011, pp. 574–583.
[15] G. Benjamin, A. Nikolaus, R. Angelika, and K. Alfons, "Load balancing in mapreduce based on scalable cardinality estimates," in *Proc. Int. Conf. Data Eng.*, 2012, pp. 522–533.
[16] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: Mitigating skew in mapreduce applications," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2012, pp. 25–36.
[17] Z. Xiao, W. Song, and Q. Chen, "Dynamic resource allocation using virtual machines for cloud computing environment," *IEEE Trans. Parallel Distrib. Syst.)*, vol. 24, no. 6, pp. 1107–1117, Jun. 2013.

[18] Amazon elastic compute cloud (EC2) [Online]. Available: http://aws.amazon.com/ec2/, 2013.
[19] L. Jimmy, "The curse of zipf and limits to parallelization: A look at the stragglers problem in mapreduce," in *Proc. 7th Workshop Large-Scale Distrib. Syst. Inf. Retrieval*, 2009, pp. 57–62.
[20] R. P. Mount, "The office of science data-management challenge," Dept. Energy, Tech. Rep. SLAC-R-782, 2004.
[21] A. Szalay and J. Gray, "2020 computing: Science in an exponential world," *Nature*, vol. 440, pp. 413–414, 2006.
[22] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proc. USENIX Conf. Oper. Syst. Des. Implementation*, 2010, pp. 1–16.
[23] A. Okcan and M. Riedewald, "Processing theta-joins using mapreduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2011, pp. 949–960.
[24] M. C. Schatz, "Cloudburst: Highly sensitive read mapping with mapreduce," *Bioinformatics*, vol. 25, no. 11, pp. 1363–1369, 2009.
[25] Z. Xiao, Q. Chen, and H. Luo, "Automatic scaling of internet applications for cloud computing services," *IEEE Trans. Comput.*, vol. 63, no. 5, pp. 1111–1123, May 2014.
[26] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proc. USENIX Conf. Oper. Syst. Des. Implementation*, 2008, pp. 29–42.
[27] Q. Chen, C. Liu, and Z. Xiao, "Improving mapreduce performance using smart speculative execution strategy," *IEEE Trans. Comput.*, vol. 63, no. 4, pp. 954–967, Apr. 2014.
[28] Open stack cloud operating system [Online]. Available: http://www.openstack.org/, 2013.
[29] K. Avi, K. Yaniv, L. Dor, L. Uri, and L. Anthony, "Kvm : The linux virtual machine monitor," in *Proc. Linux Symp.*, 2007, vol. 1, pp. 225–230.
[30] L. A. Adamic and B. A. Huberman, "Zipf's Law and the Internet," *Glottometrics*, vol. 3, pp. 143–150, 2002.
[31] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2008, pp. 1099–1110.
[32] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava, "Building a high-level dataflow system on top of map-reduce: The pig experience," *Proc. VLDB Endowment*, vol. 2, no. 2, pp. 1414–1425, 2009.



Qi Chen received the bachelor's degree from Peking University, Beijing, China, in 2010. She is currently working toward the PhD degree at Peking University. Her current research interest includes cloud computing and parallel computing.

Jinyu Yao received the bachelor's degree from Peking University, Beijing, China, in 2010. He is currently working toward the Masters degree in School of Electronics Engineering and Computer Science at Peking University. His current research interest includes cloud computing and parallel computing.

Zhen Xiao received the PhD degree from Cornell University, Ithaca, NY, in January 2001. He is currently a professor in the Department of Computer Science at Peking University, Beijing, China. After that he joined as a senior technical staff member at AT&T Labs, Middletown, NJ, and then a research staff member at IBM T.J. Watson Research Center. His current research interests include cloud computing, virtualization, and various distributed systems issues. He is a senior member of the ACM and the IEEE.