A Data Skew Oriented Reduce Placement Algorithm Based on Sampling

Zhuo Tang, Wen Ma, Kenli Li, *Member, IEEE*, and Keqin Li, *Fellow, IEEE*

Abstract—For frequent disk I/O and large data transmissions among different racks and physical nodes, intermediate data communication has become the most important performance bottle-neck in most running Hadoop systems. This paper proposes a reduce placement algorithm called CORP to schedule related map and reduce tasks on the near nodes of clusters or racks for data locality. Because the number of keys cannot be counted until the input data are processed by map tasks, this paper applies a reservoir algorithm for sampling the input data, which can bring the distribution of keys/values closer to the overall situation of original data. Based on the distribution matrix of the intermediate results in each partition, by calculating the distance and cost matrices among the cross node communication, the related map and reduce tasks can be scheduled to relatively nearby physical nodes for data locality. We implement CORP in Hadoop 2.4.0 and evaluate its performance using three widely used benchmarks: *Sort*, *Grep*, and *Join*. In these experiments, an evaluation model is proposed for selecting the appropriate sample rates, which can comprehensively consider the importance of cost, effect, and variance in sampling. Experimental results show that CORP can not only improve the balance of reduces tasks effectively but also decreases the job execution time for the lower inner data communication. Compared with some other reduce scheduling algorithms, the average data transmission of the entire system on the core switch has been reduced substantially.

Index Terms—Data sampling, data skew, inner communication, MapReduce, reduce placement

1 Introduction

1.1 Motivation

With the rapid development of the Internet and the exponentially increasing size of data, data parallel programming models have been widely used in processing terabyte- and petabyte-intensive distributed data, such as MapReduce [1], MPI [2], and OpenMP [3]. In particular, Hadoop [4] is an open source implementation of MapReduce and is currently enjoying wide popularity; however, it still has room for improvement, such as intermediate data fault tolerance [5], data skewness [6], and localized data [7]. This paper focuses on data-locality and inter-node network traffic in Hadoop, which are critical factors in the high performance of the MapReduce framework.

In the Hadoop framework, because map tasks always output the intermediate data in the local nodes, the data should be transmitted from the map nodes to corresponding reduce nodes, which is an all-to-all communication model. The frequent disk I/O and large data transmissions have become the most significant performance bottle-neck in most running Hadoop systems, which may saturate the top-of-rack switch and inflate job execution time [8]. Cross-rack communication occurs if a mapper and a reducer reside in different racks, which is very common in the environment of current data centers [9]. Due to the limitation of the switches in clusters, the overall performance of a system is often not satisfactory when working on large data-sets [10].

For map tasks, which always start at the node with current input data, to mitigate network traffic, an effective method is to place reduce tasks near the physical nodes on which map tasks generate intermediate data used as the reduce input [11]. Because the intermediate key distribution is the determining factor for the input data distribution of reduce tasks, if the intermediate data from map tasks are distributed to reduce tasks uniformly, reduce locality and task placement are unable to optimize the all-to-all communication in Hadoop. Luckily, data skew is universally existent in all input data. Current research proves that moving tasks is more efficient than moving data in the Hadoop distributed environment [12], where data skews are widespread (some key values are significantly more frequent than others). These studies allow us to solve the cross-rack/node communication problem through reduce task placement.

Most versions of Hadoop usually employ static hash functions to partition the intermediate data. This works well only when the data are uniformly distributed, but performs poorly when the intermediate results of the input are skewed. Fig. 1 illustrates the different amount of input data for each reduce task when running the "WordCount" benchmark using 10 GB of data. For reduce tasks, partitioning skew will cause shuffle skew, in which some reduce tasks will receive more data than others. The shuffle skew problem would degrade the performance in Hadoop, because a job may be delayed by a reduce task fetching large input data.

To improve system performance in this situation, many studies have focused on the data skew mitigation and tasks

- Z. Tang, W. Ma, and K. Li are with the College of Information Science and Engineering, Hunan University, and National Supercomputing Center in Changsha, Hunan 410082, China.
E-mail: {ztang, lkl}@hnu.edu.cn, mawen1029@126.com.
- K. Li is with the College of Information Science and Engineering, Hunan University, and National Supercomputing Center in Changsha, Hunan 410082, China, and with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA. E-mail: lik@newpaltz.edu.

*Manuscript received 7 Feb. 2015; revised 9 May 2016; accepted 27 Aug. 2016. Date of publication 9 Sept. 2016; date of current version 3 Dec. 2020.
Recommended for acceptance by I. Raicu, I. Foster, Y. Zhao, and J. Wozniak.
Digital Object Identifier no. 10.1109/TCC.2016.2607738*

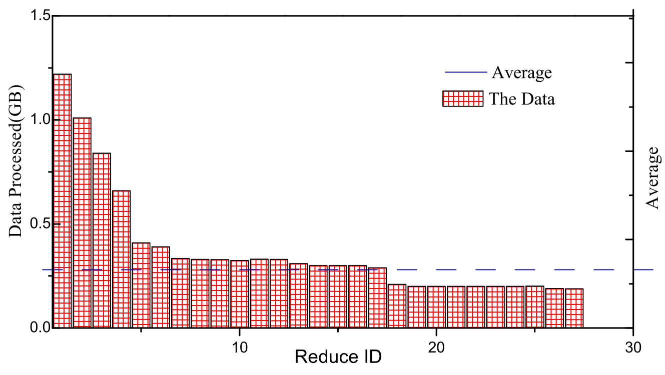


Fig. 1. Partitioning skew in reduce tasks.

load balance at present. Typically, research has addressed the problems of how to improve system performance by efficiently partitioning the intermediate keys to guarantee fair distribution of the inputs for reducers [13], improving data locality by direct task placement [7], [14], [15], or indirect task placement based on virtual machine immigration [16]. Others have attempted to improve resource utilization through speculation execution [5], [17] or task placement [15].

With these existing studies, the starting point of our work is not to solve the problems caused by data skew, but to provide a fine-grained detection method for the skewed intermediate data distribution to optimize the inner cross-racks/nodes communication through task placement. In this processing, through the calculation of the cost matrix, we consider both the data locality and the load balance.

In the Hadoop architecture, data locality and load balance are not contradictory goals, and there is no direct correlation between these two aspects. At present, many studies are attempting to optimize them synchronously. For example, Ioan et al. proposed a data-aware work stealing technique that is able to achieve good load balancing, and yet still tries to best exploit data-locality [18]. In their implementations, tasks are launched locally, but they could be migrated among schedulers for balancing loads through work stealing.

We draw many inspirations from these works. First, in the processing of a Hadoop job, the map tasks will begin at the node with its necessary input data. Although data locality is very important for map tasks, data unbalance can also damage the locality because individual node managers cannot afford excessive map tasks. Second, data skew will cause the imbalance among different reducers because of the keys dispatching based on hash, but it is necessary to consider the problem of how to place the reduce tasks based only on this imbalance condition: how to implement the locality by starting the reduce tasks on the nodes that are the source nodes of their input data.

1.2 Our Contributions

This paper proposes a communication-oriented reduce placement (CORP) method to reduce all-to-all communications between mappers and reducers, and its basic idea is to place related map and reduce tasks on the near nodes of clusters or racks. Because data skew is difficult to solve if the input distribution is unknown, a normal thought is to examine the data before determining the partition. In a real application, the intermediate outputs can be monitored and counted only after a job begins running, but it is meaningless to obtain the key value distribution after processing all input data.

To address this problem, this paper provides a dynamic range partition method that conducts a prerun sample of the input before the real job. By integrating sampling into a small percentage of the map tasks, this paper prioritizes the execution of sampling tasks over the normal map tasks to achieve the distribution statistics. The main contributions of this paper are summarized below.

- We apply the reservoir algorithm to implement the sampling for input data, and propose an evaluation model to select the appropriate sample rate. This model can comprehensively consider the importance of cost, effect, and variance in sampling.
- We propose a novel reduce placement algorithm based on data distribution, which can schedule the related map and reduce tasks on the near nodes for data locality. This algorithm can reduce the all-to-all communication among inner Hadoop clusters.
- We implement CORP in Hadoop 2.4.0 and evaluate its performance for some of the most common benchmarks. Experiment results show that CORP reduce the data transmission on core switch significantly compared with the default hash mechanism.

The rest of the paper is organized as follows. Section 2 surveys related works on reducer placement and data skew. Section 3 introduces the overall system framework. Section 4 proposes the data sampling algorithm of the MapReduce framework. Section 5 proposes the reduce placement algorithm. The performance evaluation is given in Section 6. Section 7 concludes the paper.

2 Related Works

To optimize the performance in the Hadoop framework, many algorithms and models for reduce task scheduling have been proposed in recent years. Through analysis of the current MapReduce scheduling mechanism, our early work illustrated the reasons for system slot resource wasting, which results in starvation of reduce tasks. We proposed a self-adaptive reduce task scheduling model (SARS) for the start time of reduce tasks [19]. Without the space deployment ability, SARS just determine the start time point of each reduce task dynamically according to the predicted completion time and current size of the map output. That is, SARS cannot improve the data locality and lighten the network loads in the Hadoop cluster.

Data skew is not a new problem specific to MapReduce. For the typically skewed distribution of intermediate data in Hadoop, we must face many real world applications exhibiting significant data skew, including scientific applications [17], [20]; distributed database operations such as join, grouping and aggregation [21]; search engine applications (Page Rank, Inverted Index, etc.) and some simple applications (sort, grep, etc.) [4]. Methods by which to handle data-skew effects have been studied previously in parallel database researches [22], [23], [24], but there is still no effective prediction model for the distribution of the intermediate keys.

The following studies are more similar to our works. Ibrahim et al. [13] developed a novel algorithm named LEEN for locality-aware and fairness-aware key partitioning in

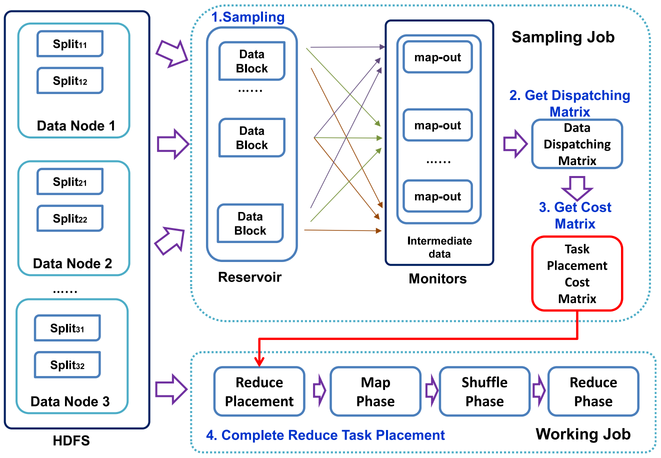


Fig. 2. The framework of CORP.

MapReduce. LEEN embraces asynchronous map and reduce schemes. All buffered intermediate keys are partitioned according to their frequencies and the fairness of the expected data distribution after the shuffle phase. However, it lacks preprocessing to estimate the data distribution effectively, so this proposed mechanism may incur significant time cost to scan the table of keys frequencies, which is generated after map tasks. This aspect of time costs is not considered in their experiments.

SkewTune is a partition algorithm proposed by Kwon et al. [17], which can remit the skew of intermediate data by redistricting the larger partition of output data from map tasks. This method is similar to the improvement of the Range algorithm [25], which is also used as a comparable algorithm in our experiments. SkewTune cannot pinpoint or split exact large keys because it does not sample any key frequency information. Therefore, as long as large keys are gathered and processed, the system cannot rearrange them. Its reduce skew mitigation cannot improve copy and sort phases, which cause performance bottleneck for some applications.

The main advantage of the works by Gufler et al. [26] is TopCluster, a proposed distributed monitoring system for capturing data skew in MapReduce systems, that can provide the cost estimation for each intermediate partition from map tasks. Hence, the partitions are distributed to the reducers such that the work load per reducer is balanced. Because the concern of this work is partition processing rather than tasks placement, the problem of data locality cannot be solved well under this model, Similar to SkewTune [17].

Tan et al. [15] formulated a stochastic optimization framework to improve the data locality for reduce tasks, with the optimal placement policy exhibiting a threshold-based structure. Their other work implemented a resource-aware scheduler for Hadoop [27] that couples the progress of map tasks and reduce tasks, utilizing wait scheduling for reduce tasks and random peeking scheduling for map tasks to jointly optimize the task placement. These excellent works are based on improving the utility of slots resources, which can improve the data locality but without enough consideration of the load balance. In addition, it is also difficult to apply in current versions of Hadoop with Yarn resource management components.

Chen et al. [6] presented LIBRA, a lightweight strategy to solve the data skew problem for reduce-side applications in MapReduce. LIBRA estimates the intermediate data distribution by sampling the partial map tasks, and uses an innovative approach to balance the load among the reduce tasks, which supports the split of large keys. Their solutions can reduce the overhead while estimating the reducers workload, but these solutions still have to wait for the completion of all the map tasks for whole input data. In addition, because data sampling is always an additional step of work for running jobs, it inevitably incurs extra running times and degrades the overall system performance.

In conclusion, there are still some problems that are not solved perfectly in these previous studies: (1) how to detect the intermediate data distribution efficiently full scanning in Hadoop job processing seems a bit ineffective; and (2) how to implement fine-grained control for the task placement: it should be modelled and quantified by an accurate cost evaluation model in the runtime environment.

3 System Overview

Our standpoint is that if map tasks and corresponding reduce tasks are placed close to each other (on the same server, same rack, etc.), the system would cost less for the same amount of traffic relative to a case with the reduce tasks located far from the node. Furthermore, in addition to improving the performance of an application, minimizing the communication cost will also reduce the network overhead of the underlying infrastructure by moving traffic from bottleneck links to high-bandwidth links.

Fig. 2 shows the overall execution produce, which is composed of two separate jobs. First, the original input data are sampled to estimate the source of key/value tuples for each reducer by nodes. The output of this phase is a matrix to record the size and key/value distribution of current input data, which can be transferred to a cost matrix for reduce task placement. Before the working job is run, the reduce task placement should be finished according to this matrix. That way, most data handled by a reduce task handles be localized as much as possible, thus saving traffic cost and improving the performance of the reduce tasks. The main steps are as follows.

Data Sampling: Input data are loaded into a file or files in a distributed file system (DFS) where each file is partitioned into smaller chunks, called input splits. Each split is assigned to a map task. Map tasks process splits, and produce intermediate outputs which are usually partitioned or hashed to one or many reduce tasks. Before a MapReduce computation begins with a map phase, in which each input split is processed in parallel, a random sample of the required size will be produced. The splits of samples are submitted to the auditor group, while the master and map tasks wait for the results of the auditor.

Reduce Task Placement: The results of sampling will determine the placement of reduce tasks. Fig. 3 briefly shows a typical example. For 80 percent key/value pairs, reduce task R_1 comes from map task M_2, and the remaining intermediate results are from map task M_1. Hence, the most appropriate position to start the task of reduce task R_1 is the node on which map task M_2 is running. Analogously, to

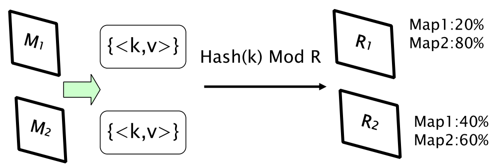


Fig. 3. The intermediate results distribution in reduce tasks.

obtain better data locality and save the inner communication among nodes, it is better to launch reduce task R_2 on the node of map task M_1.

4 Data Skew and Data Sampling in MapReduce Framework

4.1 Data Skew Model

In this model, to quantify the data received by a special reduce task, some initial and intermediate results with their relationships can be formalized as Table 1.

$C = C_{i,j}^{\sigma,l}$ is a three-dimensional matrix of $m \times p \times n$ that defines the distribution of intermediate results in each partition. $C_{i,j}^{\sigma,l}$ denotes the number key/value tuples processed by the jth reducer from the ith map task within the lth node. And $C_{i,j}^{\sigma,l} = k$ means that k pairs of keys/values from map task M_i in node N_l are currently allocated to reduce task j. For a partition that represents the tuple set processed by the same reducer, the number of partitions is treated equally to the reducer amount. In this model, n denotes the number of nodes, p denotes the number of reducers, and m denotes the number of map tasks. Hence, $0 \leq l < n$, $0 \leq j < p$, and $0 \leq i < m$.

Under normal conditions, the key number of original input data follows a Zipf distributions [28]. Parameter σ is used to denote the degree of the skew, which is usually assigned from 0.1 to 1.2. For a specific input dataset, parameter σ is a constant. A larger value indicates heavier skew, and it also determines the distribution of $C_{i,j}$. In this model, the number of key/value pairs processed by the reducer j is denoted as $RC(j)$. Without loss of generality, the value of $RC(j)$ with a skew degree could be defined as follows:

$$RC(j,\sigma) = \sum_{l=0}^{n-1} C_j^{\sigma,l} = \sum_{l=0}^{n-1}\sum_{i=0}^{m-1} C_{i,j}^{\sigma,l}. \tag{1}$$

On this basis, we can calculate the average number of key/value tuples of all running reduce tasks as

$$\overline{mean_\sigma} = \frac{\sum_{j=0}^{p-1} RC(j,\sigma)}{p} = \frac{\sum_{j=0}^{p-1}\sum_{l=0}^{n-1}\sum_{i=0}^{m-1} C_{i,j}^{\sigma,l}}{p}, \tag{2}$$

in which parameter p is the number of reduce tasks.

Naturally, the intermediate data processed by a reduce task can be considered as skew using standard deviation as

$$|RC(j,\sigma) - \overline{mean_\sigma}| > std, \tag{3}$$

std is the standard deviation of the number of key/value tuples for all reduce tasks, which can be used to measure the overall load balancing level of reducers.

TABLE 1
Variable Declaration

$n, 0 \leq l < n$	n: node number; l: one node;
$p, 0 \leq j < p$	p: reducer number; j: one reducer;
$m, 0 \leq i < m$	m: mapper number; i: one mapper;
$C_{i,j}^{\sigma,l}$	key/value numbers from the mapper M_i in node N_l received by jth reducer;
$RC(j)$	number of key/value pairs processed by reducer j;
$\overline{mean_\sigma}$	average number of key/value tuples of all running reduce tasks;
std	the standard deviation for the current loading of reducer;
FoS	an indicator to measure the load balance of reducer.

Further, we can evaluate the difference between the average intermediate results of all reduce tasks and the number of key/value pairs belonging to the jth reducer as

$$(RC(j,\sigma) - \overline{mean_\sigma}) = \sum_{l=0}^{n-1}\sum_{i=0}^{m-1} C_{i,j}^{\sigma,l} - \frac{\sum_{j=0}^{p-1}\sum_{l=0}^{n-1}\sum_{i=0}^{m-1} C_{i,j}^{\sigma,l}}{p}. \tag{4}$$

The value of this standard deviation for all intermediate results in reduce tasks can be calculated as

$$\begin{aligned} std[(RC(j,\sigma)] &= \sqrt{\frac{\sum_{j=0}^{p-1}(RC(j,\sigma) - \overline{mean_\sigma})^2}{p}} \\ &= \sqrt{\frac{\sum_{j=0}^{p-1}\left(\sum_{l=0}^{n-1}\sum_{i=0}^{m-1} C_{i,j}^{\sigma,l} - \frac{\sum_{j=0}^{p-1}\sum_{l=0}^{n-1}\sum_{i=0}^{m-1} C_{i,j}^{\sigma,l}}{p}\right)^2}{p}}. \end{aligned} \tag{5}$$

In this case, when a reduce task is load balanced, $|RC(j,\sigma) - \overline{mean_\sigma}| < std$ is always satisfied. As a result, when the number of key/value tuples assigned to reducer j is larger than the value of the mean, the jth reducer will be taken as a skew task even although it is running normally.

To measure the data skew degree of all reduce tasks, this paper uses the indicator FoS (Factor of Skew) to quantize data skew and load balancing

$$FoS = std[(RC(j,\sigma)]/\overline{mean_\sigma}. \tag{6}$$

The smaller the value of FoS, the better the load balancing and the lower data skew that will be obtained.

4.2 Data Sampling Algorithm

To ascertain the distribution of the intermediate data is the only way to develop a reduce placement strategy. Because the number of keys cannot be counted until the input data are processed by map tasks, calculating the optimal solution to the above problem is unrealistic, and the cost of pre-scanning the whole dataset would likely be unacceptable when the amount of data is huge. Therefore, we present a distributed approximation algorithm by sampling and estimation.

In most running Hadoop systems, sampling of the input data can be achieved by using the class: org. apache. hadoop. mapred. lib. InputSampler. This class implements the SplitSampler method which samples records from the first S partitions. The original release is a random sampling of data convenient way. Reading a record will be added to the sample set in each partition as long as the current number of samples is less than samples needed, by means of looping through all records in this partition. Our sample strategies are also run by invoking this InputSampler class in the underlying implementation. To improve the initial random selection strategy, this paper overloads the SplitSampler method and proposes a more efficient selection model for sample data based on reservoir sampling.

Conventional uniform sampling will inevitably result in a certain number of multi-duplicated samples. Because all random number generators in the Java and Scala languages are simply pseudorandom functions, for large-scale data, especially with increasing sampling space, they cannot guarantee that all sample data are completely randomized. From [29], the main process of reservoir sampling is to save k preceding elements first (k is the sample number and also the size of the reservoir) and then randomly replace original selected elements in the reservoir using a new element that is selected from outside the reservoir in a different probability. The final k sample data will be generated after finishing the traversal for current input data. Compared with pseudorandom functions, reservoir sampling can ensure randomness, especially when taking the data from some sequence flows, and it is ideal for reading the input data from large texts line by line in the Hadoop/Spark framework. Compared with conventional uniform sampling, reservoir sampling can ensure that the key distributions are closer to the whole situation in the original data.

Algorithm 1 provides the process to obtain the distribution of the intermediate tuples for each reduce task. For a specific MapReduce job, this algorithm first starts this job with the sample data and records the number of tuples from a map local node to each reduce task based on a monitor in each map node. As a practice, a data cluster is the subset of all pairs with the same key in this paper, and all clusters processed by the same reducer constitute a partition [6]. The function $getOrignalMapNode$ is used to retrace the intermediate tuples and obtain the map nodes that produce these data [30]. From Algorithm 1, because the intermediate tuple distribution of sample data remains coherent with the whole input dataset, we can calculate the data size of reduce tasks from every map node under the consistent distribution law.

Obviously, there is a trade-off between the sampling overhead and the accuracy of the result. The experiments in Section 6.2.1 are designed to select an appropriate sample rate that satisfies these seemingly contradictory necessities. However, it is important to note that not all jobs must sample the data before running. The key distribution is the objective, which is indeed application independent. For jobs with the same input, after uploading the original data to the Hadoop Distributed File System (HDFS), their original storage distributions in data nodes are relatively fixed.

For the matrix $C = C_{i,j}^{\sigma,l}$, the number and position of map tasks on the nodes are simply up to the sizes and distributions of the input data, which can determine subscripts i and l for the element in matrix C. Subscript j is just the order number of the reduce tasks, but the number of reduce task can actually be pre-set in the source code. Hence, for this situation, we must perform data sampling only once, and the obtained matrix C can be reused by different jobs.

Algorithm 1. Distribution Detection

```
Input:
  The sample of data blocks $BS$;
  a MapReduce job: $mr_j$;
  the number of computing nodes $N$;
  the number of reduce tasks $R$;
Output:
  The matrix $C$ of whole input data.
  run the MapReduce job $mr_j$ using $BS$ as the input data;
  initialize an intermediate matrix $C$.
  $C_{i,j}^{\sigma,l} \leftarrow 0, 0 \leq l < N, 0 \leq j < R$;
  for each cluster $c_k$ with tuple key $k$ do
      //get reducer serial number for $c_k$;
      $j \leftarrow systemhash(k)$;
      for each map node $l$ do
        for each map task $i$ do
          //if $c_k$ is come from node $l$
          if $l$=$getOrignalMapNode(c_k)$ then
            $C_{i,j}^{\sigma,l} \leftarrow C_{i,j}^{\sigma,l} + tuple\_number\_of(c_k)$;
          end if
        end for
      end for
  end for
  $SN \leftarrow$ the number of tuples in $BS$;
  $WN \leftarrow$ the number of tuples in whole input data;
  for each map node $l$ do
      for each map task $i$ do
        for each reduce task $j$ do
          $C_{i,j}^{\sigma,l} \leftarrow (C_{i,j}^{\sigma,l} \times WN) / SN$;
        end for
      end for
  end for
  return matrix $C = \left\{C_{i,j}^{\sigma,l}\right\}$.
```

5 Communication Oriented Reduce Placement

5.1 The Model of MapReduce

In most implementations of various Hadoop versions, the key-value pairs space is partitioned among the reducers. The partitioner design has a direct impact on the overall performance of the job: a poorly designed partitioning function will not evenly distribute the load over the reducers. As the default partitioner, HashPartitioner hashes a record key to determine the partition (and thus which reducer) in which the record belongs. The number of partitions is then equal to the number of reduce tasks for the job [31].

Let n be the number of nodes, and m be the number of map tasks. We first initialize the node set as $\{N_0, N_1, \ldots, N_l, \ldots, N_{n-1}\}$, and the map set as $\{M_0, M_1, \ldots, M_i, \ldots, M_{m-1}\}$, $0 \leq n \leq m$ for rack set $R_r, r \in \{0, 1, \ldots, k-1\}, 0 \leq k < n$. For this model, some specific data structures can be formalized as follows:

(1) V: A vector of length p whose elements indicate the relevant number of key/value tuples in every node.

If $v_{l,j} = k$, there are k key/value pairs in node N_l assigned to reduce j. Therefore,we have

$$v_{l,j} = \sum_{i=0}^{m-1} C^l_{i,j}, 0 \le l < n, \tag{7}$$

and

$$V_l = [v_{l,0}, v_{l,1}, \ldots, v_{l,p-1}], C = [V_0, V_1, \ldots, V_{n-1}]. \tag{8}$$

(2) D: A matrix of $n \times n$ that defines the distance between physical nodes. According to the network latency, we can define the distance between two physical nodes in the same rack as d_1, the distance between two physical nodes in different racks but in the same cluster as d_2, the distance between two physical nodes in different clusters as d_3, and the distance between two physical nodes in different data centers as d_4. The distance among maps is the distance between the two nodes in which the maps are located; i.e., the distance among maps in the same node is 0. In the four situations above, the distance value would increase with increasing physical distance: $0 < d_1 < d_2 < d_3 < d_4$. This paper supposes that the shorter the distance, the faster the data transfer speed, which is the theoretical basis of the model optimization.

(3) R: A matrix of $p \times n$ that defines the position of reduce tasks started on the node, and is a typical sparse matrix. The element $r_{l,j}$ is a Boolean variable that indicates whether reduce task j is set up on node l. The following condition ensures that each reduce task should be placed on only one node:

$$\sum_l r_{l,j} = 1, \forall l \in [0, n-1]. \tag{9}$$

The quantity of reduce tasks is usually less than that of map tasks. A reduce task can be started in a physical node only if the physical node can provide sufficient computing resources apart from the existing map tasks. In this model, the allocation matrix R indicates the position at which to start up the reduce tasks, meanwhile, another matrix D defines the mutual distances between the physical nodes. On this basis, this model uses a communication matrix T among nodes to quantify the cost of data transmission called intermediate results which are copied from map tasks to reduce tasks in cluster. On this basis, we use a vector RV_j to denote the position of a task on a node in the matrix R:

$$RV_j = [r_{0,j}, r_{1,j}, \ldots, r_{l,j}, \ldots, r_{n-1,j}]. \tag{10}$$

The element $r_{l,j}$ in vector RV_j is a Boolean variable that indicates whether reduce task j is placed on node l. Therefore, if there are r reduce tasks among nodes, we can define matrix R as follows:

$$R = [RV_0, RV_1, \ldots, RV_{r-1}]^T. \tag{11}$$

To capture locality, we define a communication cost function that measures the data transfer between two nodes. In most Hadoop implementations, the largest cost of a MapReduce application is in copying inner result data from map tasks to relevant reduce tasks that are placed on the other physical nodes. Using the network hops as the near-far measure, this paper defines a vector DV to calculate the distance among node N_l and other nodes:

$$\begin{aligned} DV_l = [dis_{l,0}, dis_{l,2}, \ldots, dis_{l,d}, \ldots, dis_{l,n-1}] \\ 0 < |dis_{l,d}| \le d_4, 0 \le l, d < n. \end{aligned} \tag{12}$$

For the process of map tasks outputting the intermediate key-value set in their local node and the related reduce tasks of fetching the corresponding data through the network, there are two key factors regarding cost matrix T: the amount of data transferred and the distances between the map nodes and reduce nodes. An element $t_{l,j}$ of the cost matrix can be obtained as follows:

$$\begin{aligned} t_{l,j} &= DV_l \times RV_j^T, \forall l \in [0, n-1], \forall j \in [0, p-1] \\ &= \sum_{d=0}^{n-1} dis_{l,d} \times r_{d,j}. \end{aligned} \tag{13}$$

In the foregoing discussion, C^l is defined as an intermediate result allocation matrix, which represents the key/value pairs distribution in node l. $t^k_{l,j}$ denotes the kth placement choice for reduce task j on node l. With cost matrix T, the minimal cost MC of the whole Hadoop system can be calculated by multiplying the distribution matrix C by the cost matrix T from a distinguished central node:

$$MC\left(C^{\sigma,l}, T\right) = \min_k \left(\sum_{i=0}^{m-1} \left[\sum_{j=0}^{p-1} c^{\sigma,l}_{i,j} \right] \times t^k_{l,j} \right). \tag{14}$$

From Eq. (14), in the data skew environment, if most data for reduce task j come from node l, when MC obtains the minimal value, the jth reduce task can fetch the largest local data blocks in the kth computing node. Here, m is the number of the map tasks, and p is the number of partitions. To obtain the load balance of all reduce tasks, Algorithm 2 is designed to combine the smaller clusters of $<key, value>$ tuples to an optimal reduce task, according to the current workload of the reduce tasks. Algorithm 2 adjusts the intermediate data distribution matrix C which is obtained from Algorithm 1 before the system runs. The measure of the FoS value for typical benchmarks in the experiments also verifies that this algorithm can maintain the load balance for M/R tasks effectively.

To achieve an optimal reduce task placement solution that can minimize the network communication overhead among different physical nodes, the objective function of this model can be specified as the following optimization problem:

$$Minimize \sum_{m,n} MC(C^{\sigma,l}, T), \tag{15}$$

where n denotes the number of nodes. Hence, the reduce task placement can be specialized as a problem to obtain the assignment of matrix T, which can achieve the target of Eq. (15). The Cost Matrix Generation algorithm shows the specific steps to calculate cost matrix T by multiplying distribution C by distance matrix D.

Algorithm 2. Cluster Combination

```
Input:
  A collection of tuple set: $TS = \{ts_1, \ldots, ts_k, \ldots, ts_K\}$;
  the collection of reduce tasks: $R = \{r_1, r_2, \ldots, r_j, \ldots, r_p\}, p \leq K$;
  the predicted matrix $C$ of whole input data from Algorithm 1.
Output:
  The adjusted matrix $C$ of input data.
calculate the size $|ts_i|$ of each cluster. $1 \leq i \leq K$;
sort $TS$ in descending order according to $|ts_k|$;
assign $p$ selected clusters from $TS$ sequentially to $p$ reducers;
for each $k \in [p+1, K]$ do
  for each map task $i$ do
    assign cluster $ts_k$ to reducer $r_p$;
    //get the map node for a specific reduce task;
    $l \leftarrow getOrignalMapNode(ts_k)$;
    $j \leftarrow p$;
    //adjust matrix $C$ as output;
    $C^l_{i,j} \leftarrow tuple\_number\_of(ts_k)$;
    sort $R$ in descending order;
  end for
end for
return matrix $C$.
```

Algorithm 3. Cost Matrix Generation

```
Input:
  $n$: the number of node;
  $C$: split-partition matrix;
  $D$: distance of resources.
Output:
  The cost matrix $T$.
$partitionList \leftarrow \emptyset$;
//traverse each element in matrix $C$;
for each $l \in [0, n-1]$ do
  $PartitionList = getPartitionsList(C, i)$;
  for $j$ in $partitionList$ do
    //Count pairs in every partition at node $l$;
    $tempC[j] \leftarrow Calculate(C, j, D)$;
  end for
  $tempT \leftarrow tempC \times getDist(D, j)$;
  $T \leftarrow Com(tempQ)$;
  continue;
end for
return $T$.
```

5.2 Placement Algorithm for Reduce Tasks

In Algorithm 3, the $getPartiionsList()$ method returns list partitions on the lth node.

Parameter D is the distance matrix. The method $Calculate()$ returns the value with the number of intermediates in node n_l, and $Com()$ returns the minimum value of the array.

In Algorithm 4, the $getNodesList()$ method returns the list of node, and $getCost()$ will obtain the cost of placing the jth reduce task on the lth node. The minimum value can be selected in the array by the method $Minimize()$.

6 Experimental Evaluation

The experiments in this paper are divided into two steps. First, we provide a detailed micro-benchmarking for our data and put forward reduce placement techniques for each MapReduce job class on a real cluster composed of 20 physical machines. Second we present a detailed and quantitative analysis of the result for mixed job types executed under this reduce placement algorithm.

Algorithm 4. Reduce Task Placement

```
Input:
  $p$: the number of reduce;
  $n$: the number of node;
  $T$: the cost matrix.
Output:
  The placement queue $R$ which able to minimize the value of
  $MC$.
List $nodesList \leftarrow \emptyset$;
for each $j \in [0, p-1]$ do
  $nodesList \leftarrow getNodesList(N, T)$;
  for $l$ in nodes list do
    //obtain the cost on $l$th node;
    $tempSum[l] \leftarrow getCost(T, j, l)$;
  end for
  $R[l] \leftarrow Minimize(tempSum)$;
end for
return $R$.
```

6.1 Experiment Setting

In our experiments, the following reduce scheduling algorithms are chosen for comparison.

NorP (Normal Placement). In original Hadoop implementations, reduce tasks are launched according to random assignment and the resource utilization in the computing nodes. In Hadoop version 2.0 or higher, this distribution can be controlled by the programmer in the YARN framework, which is the implementation mechanisms of CORP: to complete the reduce placement by invoking these YARN APIs [4].

Range (Range Partition). This is a widely used algorithm of partition distribution. In this method, the intermediate $<key, value>$ tuples are first sorted by key, and then the tuples are sequentially assigned to the reduce task according to this key range. Because the partitions may be split or combined in to different reduce tasks, this algorithm can ensure the furthest load balance of the reduce tasks [6], [25].

Self-Adaptive Reduce Scheduling. This is an optimal reduce scheduling policy for reduce tasks start time in the Hadoop platform [19]. It can decide the start time points of each reduce task dynamically according to each job context, including the task completion time and the size of the map output. This model can decrease the reduce completion time and the system average response time in the Hadoop platform effectively [19].

As shown in Fig. 4, our cluster consists of 20 physical machines loaded with the operating system of Ubuntu 12.04 (KVM as the hypervisor) with 16 core 2.53 GHz Intel processors. Those machines are managed by CloudStack, which is an open source cloud platform with two racks, each of which contains 10 physical machines. The network bandwidth is 1 Gbps, and the nodes within a rack are connected through a single switch. In these experiments, the volume of intermediate data transmissions of the whole system on

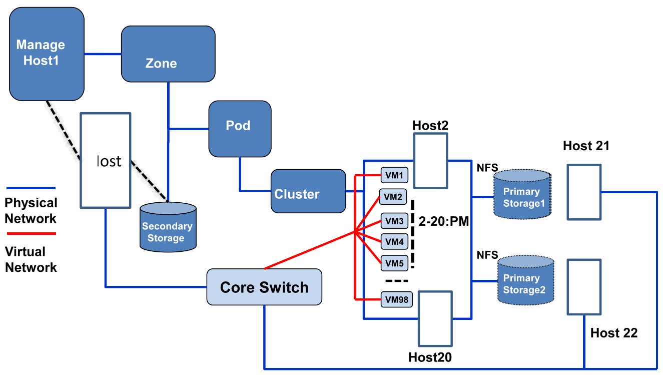


Fig. 4. Experiment network topology.

this core switch can be counted by the monitor of the network management software. Each job uses a cluster of 50 VMs with each VM configured with 4 GB of memory and four 2 GHz vCPUs. A description of the various job types and the dataset sizes are shown in Table 2.

6.2 Performance Evaluation

6.2.1 Sampling Experiments

In this section, we first propose an evaluation formula as Eq. (16) to select the appropriate sample rate, which can comprehensively consider the importance of cost, effect, and variance in sampling

$$i = argMin[f_i(\Delta_i, T_i, \Phi_i) = \alpha\Delta_i + \beta T_i + \gamma\Phi_i], \tag{16}$$

where function $f_i(\Delta_i, T_i, \Phi_i)$ is a comprehensive index considering both cost and effect, in which Δ_i reflects the difference among the sequences of FoS values between the currently adopted percentage and 100 percent (the whole input dataset), which can be calculated as

$$\Delta_i = \sqrt{\sum_{j=1}^{N}\left(d_{i,j} - \frac{1}{N}\sum_{j'=1}^{N} d_{5,j'}\right)^2}, \tag{17}$$

where N denotes the experimental repetition time, and $d_{i,j}$ represents the FoS value obtained in the jth sampling experiment under the ith sampling rate. $1 \leq i \leq SN$ is the order number of different sampling percentages: {1 percent, 25 percent, 50 percent, 75 percent, 100 percent}, and SN denotes the space size of different sampling rates. For this experiment, $SN = 5$, and $d_{5,j}$ denotes the values with a 100 percent sampling rate.

As an average sampling execution time, T_i can be calculated simply as

$$T_i = \frac{1}{N}\sum_{j=1}^{N} t_{i,j}, \tag{18}$$

TABLE 2
Job Types and the Dataset Sizes

Workload classification	Benchmarks	Input data
Map and Reduce-input heavy	sort	10 G
Map-input heavy	Grep:word search	10 G
Reduce-input heavy	Join	2 G

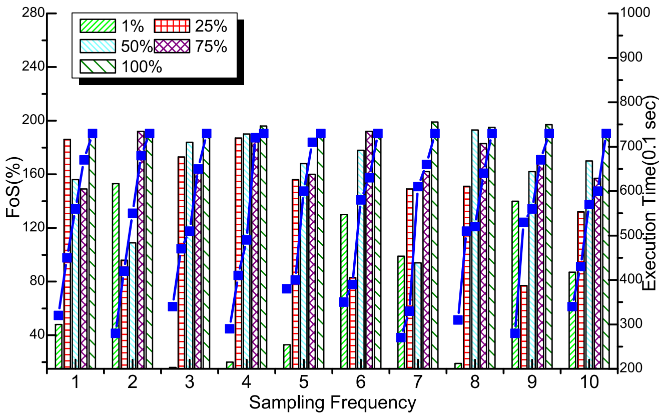


Fig. 5. The comparison experiment with various sampling rate.

where $t_{i,j}$ represents the execution time of the jth sampling experiment under the ith sampling rate, $1 \leq i \leq SN$ and $1 \leq j \leq N$.

To full consider the influence of data volatilities, Eq. (19) provides the process to calculate the parameter Φ_i based on the standard deviation formula:

$$\Phi_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N}\left(d_{i,j} - \frac{1}{N}\sum_{j=1}^{N} d_{i,j}\right)^2}. \tag{19}$$

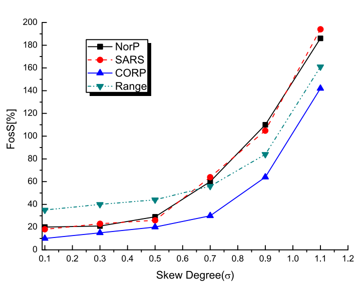
Fig. 5 shows the FoS and execution time obtained in times sampling experiments. Based on these results, Table 3 provides the final values of Δ_i, T_i, and Φ_i for all benchmarks with various sampling rates. Each group of sampling experiments is repeated ten times, which means that the parameter N in Eqs. (17), (18) and (19) should be set to 10 in these experiments. Finally, for the weight coefficients which reflect the importance of cost, effect, and variance in sampling, we can simply set $\alpha = \beta = \gamma = 1$. That is, we think that the cost, effect, and data volatility are equally important.

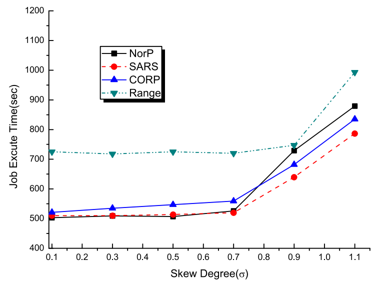
Moreover, what is noteworthy is the group of experiments with a sample rate of 1 percent: in this case, the time costs are relatively low, and most of the FoS values are lower than the other sample rates. However, it is easy to find that the experimental results of FoS values and time costs are very volatile, and the results of Δ_i in Table 3 confirm that there is great difference compared with FoS values with a 100 percent sample rate, which are actually measured by Eq. (17). We hold that the lower sample rate cannot easily represent the accurate distribution of the whole input dataset.

By comprehensive consideration according to Eq. (16), it is easy to learn that sampling 25 percent of the map tasks is

TABLE 3
Comprehensive Evaluation for Cost and Effect with Different Benchmarks

i	$rate$	Δ_i	T_i	Φ_i	f_i
1	1%	407.307	316	51.173	774.48
2	25%	209.564	434	38.833	682.397
3	50%	143.043	555	31.668	729.711
4	75%	80.188	663	14.925	758.113
5	100%	11.580	730	3.662	745.242


(a) load balance


(b) job execution time

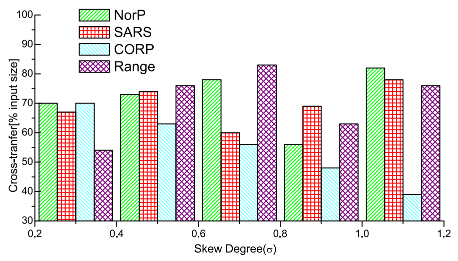

(c) inner data traffic

Fig. 6. Performance versus data skew for *Sort*.

an appropriate choice for the input data of *Sort* and *Grep* benchmarks. For benchmark join, the most appropriate sample rate is 1 percent.

As mentioned previously, the major motivation for CORP is to improve the data locality to diminish cross-rack communication. In the following experiments, to verify the advantages of CORP, we evaluate this model for FoS and job execution time using these common benchmarks: *Sort*, *Grep*, and *Join*.

As mentioned in Section 4.2.1, for the jobs with the same input, we need only to perform data sampling once, and the obtained matrix C can be reused by different jobs. Moreover, because the sampling target is merely detects the distribution of intermediate data, the time cost is much lower than the practical jobs.

6.2.2 Sort Benchmark Testing

The Sort benchmark in Hadoop is usually used for workload testing because it is a heavy job for map and reduce inputs. In these experiments, we generate 10 GB of synthetic data sets following Zipf distributions with σ parameters varying from 0.1 to 1.2 to control the degree of skew. The reason to choose Zipf to describe the frequency of the intermediate keys is that this distribution is very common in data coming from human society [26].

In the following experiments, the performance evaluations of the relevant algorithms are illustrated considering their job execution time in the change of the input data skew degree. In this paper, the load balancing and skew degree are measured by the factor of skew. Fig. 6 shows the experimental results based on the 10 GB of synthetic data. From Fig. 6a, we can conclude that CORP can improve the FoS obviously for different input data with various skew degrees. The curves in Fig. 6b show that the job execution performance of CORP remains better than normal placement but worse than SARS. Through this result, we can know that CORP can decrease the execution time in the reduce phase because it can make the intermediate results more localized.

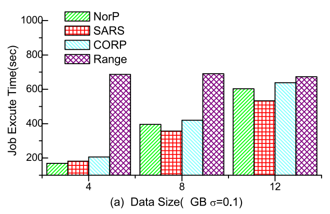
More specifically, as we can see in Fig. 6a, the value of FoS increases rapidly when the degree of skew exceeds 0.7 for all the experimental algorithms, and from Fig. 6b, the execution times of all algorithms also increase substantially once the degree of skew reaches a certain threshold. Through the reduce placement, CORP ensures even data processing in the nodes. From the results, this algorithm has better performance in terms of FoS optimization, but it performs poorly in terms of execution time.

The results in Fig. 6 shows that, by sampling to determine the placement of reduce tasks, CORP can obtain a better data load balance, lower FoS and less transfer in the cross-rack compared with other reduce scheduling algorithms. However, when the skew degree is less than 0.7, CORP performs worse than NorP and SARS in terms of execution time. The reason is that CORP must calculate the cost matrix and make the decision on reduce task placement after running a separate sampling job, which would incur significate extra overheads. However, because the data transmission among different nodes is optimized through reduce local placement in CORP, the whole execution time of a job can be decreased to offset the time spent, which is increased by the extra overhead for decision-making. For this reason, although the performance of CORP and Range are both lower among these algorithms, CORP has a much lower execution time than Range.

A similar trend is seen in Fig. 6c, in which the bars illustrate the transfer data through the core switch as a percent of overall inner data traffic. As the skew degree increases, an approximate task location produced by CORP can help balance the expected load among physical machines and increase data locality compared with other algorithms, which focus only on the load balancing without consideration given to data locality. The data transmission of CORP on the core switch have been decreased by up to 51.9 percent compared with NorP with the skew degree set to $\sigma = 1.1$ (in the Fig. 6c).

Fig. 7 represents the variation of job execution time when the data size ranges among {4 GB, 8 GB, 12 GB} with different skew degrees ($\sigma = 0.1$ and $\sigma = 1.2$). When the data set has a smaller skew degree (see Fig. 7a), SARS is the most time-efficient algorithm, and CORP performs better than only NorP owing to the overhead for data sampling and task placement decision-making.

With the increased skew degree and the data size (see Fig. 7b), the growth rate of the CORP execution time is smaller than the others. However, in these two experiments, Range has the worst execution time owing to its even splitting and combining strategy for intermediate tuples. This


(a) Data Size(GB σ=0.1)

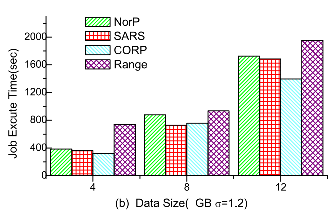

(b) Data Size(GB σ=1.2)

Fig. 7. Performance versus data size for *sort*.

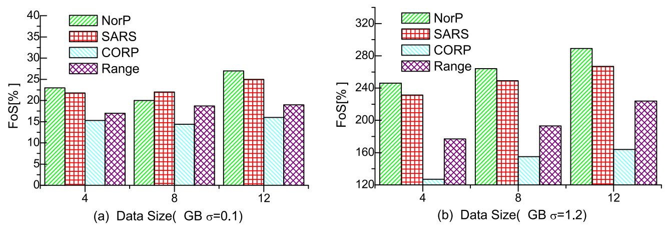


Fig. 8. FoS versus data size for *Sort*.

causes poor locality in the reduce phase and the redundant inner communication finally results in the longer execution times. Especially when the data-set size gradually approaches 12 GB, the execution time of CORP becomes smaller than SARS with skew degree $\sigma = 1.2$. The reason is that the appropriate location of reduce tasks can help achieve much greater localization, and decrease the unnecessary communication among racks. This experiment also demonstrates that the performance of CORP is relatively high when the scales of the skew degree and data set are large.

Fig. 8 illustrates the relationship between FoS and the data size: the smaller the FoS, the better the load balancing (keeping $\sigma = 1.2$). When the data set has a smaller skew degree and data size varies from 4 GB to 12 GB (see Fig. 8a), we can observe that the FoS values of all methods increase slowly, but this indicator of CORP is always better than the other compared algorithms. As the dataset scale increases rapidly, the performance of CORP is more prominent: the FoS in our system is 60 percent smaller than NorP in the Hadoop implementation.

As shown in Fig. 8b, the factor of skew among all reduce tasks in CORP is only 160 percent, whereas, it reaches 280, 260, and 230 percent in NorP, SARS, and Range, respectively.

6.2.3 Grep Benchmark Testing

Grep is a popular application for large-scale data processing with heavy map-input. It searches some regular expressions through input text files and outputs the lines that contain the matched expressions. We improve 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 data set we used is the full English Wikipedia archive with a total data size of 10 GB, which constitute the data processing jobs with heavy map tasks.

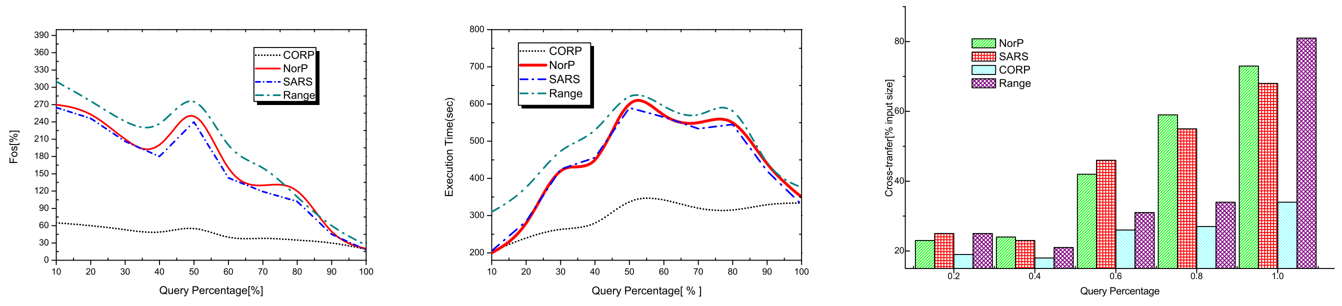
Because the behaviour of Grep depends on how frequently the search expression appears in the input files, we tune the expression and make the input query percentages vary from 10 to 100 percent. Fig. 9 shows the changes of job execution time, FoS value and cross-rack transfer with increasing query percentage. Note that most current versions of Hadoop do not provide a suitable range partition for this application: their pre-run sampler can detect the input data but cannot handle applications in which the intermediate data are in a different format from the input.

In contrast, the proposed sample algorithm in this paper can spot-check the intermediate data directly and works well for all types of applications. As we can see in Fig. 9a, CORP performs obviously better than NorP, SARS, and Range with lower query percentage. This is because CORP has obvious advantages for searching unpopular words in the archive and tends to generate results with heavy data skew. Although the curve of execution time is always lower than those of other algorithms in this experiment, as the query percentage increases, because the distribution of the result data becomes increasingly uniform, the performance gap rapidly closes. As a matter of fact, when the query percentage approaches 100 percent, as shown in the Fig. 9b, the performance of CORP is very similar to the other three algorithms.

The inner data transfer in the Fig. 9c shows that CORP has lower traffic (the highest is only 30.1 percent) compared with the NorP, SARS, and Range algorithms for various query percentages.

6.2.4 Join Benchmark Testing

Join is one of the most common reduce-input heavy applications, especially in a data skew environment. We implement a simple broadcast Join job in Hadoop that partitions a large table in the map phase, whereas a small table is directly read in the reduce phase to generate a hash table to speed up the Join operation. When the small table is too large to fit into the memory, we use a buffer to maintain only a part of the small table in memory and use the cache replacement strategy to update the buffer. For this experiment, we select a data set with size = 2 GB and $\sigma = 1.2$ from the widely used corpora *"Yahoo News Feed dataset, version 1.0"* [32] to evaluate the time performance and load balance effect. CORP is compared with other algorithms under Hash Join (PHJ) and Replicated Join (PRJ) in Pig [33]. Fig. 10a shows the load balance and job execution time of these three test cases. In Fig. 10a, 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.



(a) load balance (b) job execution time (c) inner data traffic

Fig. 9. Performance versus query percentage for *Grep*.

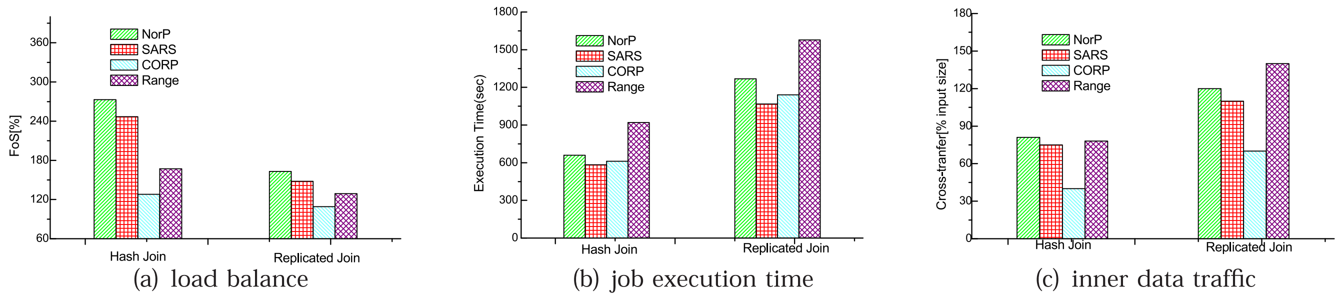


Fig. 10. Performance versus Hash and Replicate for *Join*.

In Fig. 10b, the best scheme in Pig is PHJ, which samples a large table to generate the key distribution and makes the partition decision beforehand. In Fig. 10c, the difference in job execution time can be explained by inner data network traffic. In CORP, the data can be read from closer nodes, and lower time cost of communication can decrease the whole job execution time. However, because the data sampling will be additional works that can incur extra run time, the job execution speed in CORP is certainly lower than that of SARS.

6.2.5 Experiment Summary

In the foregoing experiments, we evaluate the performance and effect using input data with even variable skew degrees. To verify the effectiveness of synthesized data for *Sort* and *Join*, a group of comparative experiments are designed with the results shown in Fig. 11. For the chosen realistic data, we generate some different sizes of synthesized data with similar skew degree and repetition of keys. In this example, $\sigma = 0.7$. From Fig. 11, we can easily see that the experimental results of *FoS* and execution time on synthesized data are roughly identical to the real loads.

The squares in Fig. 12 summarize the comparisons among these algorithms: NorP, SARS, CORP, and Range. The results in Fig. 12a show that CORP can achieve the shortest execution time for the smallest data communication from map tasks to reduce tasks. The comparison values of FoS in Fig. 12b further illustrate that, benefiting from the data-locality, CORP can achieve better load balance than other algorithms within the workloads: *Grep*, *Sort*, and *Join*.

The final group of experiments is to test the performance of batch jobs. Because one job client always waits for the execution to complete after the job is submitted, our way is to submit various numbers of jobs using multiple shell clients at the same time. The execution time is the duration from the start of the first job to the end of the last job. Fig. 13 records the execution time under CORP for batch jobs using the benchmarks: *Sort*, *Join*, and *Grep*. It is easy to see that the time cost always sharply increases with more than 6 concurrent jobs.

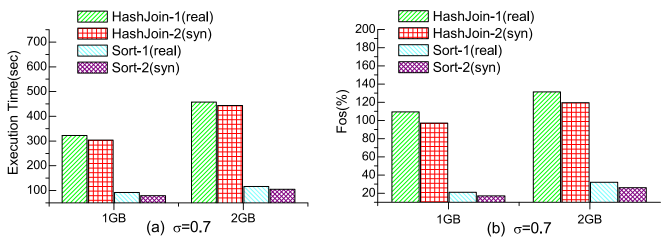


Fig. 11. Effect evaluation for real and synthesis data.

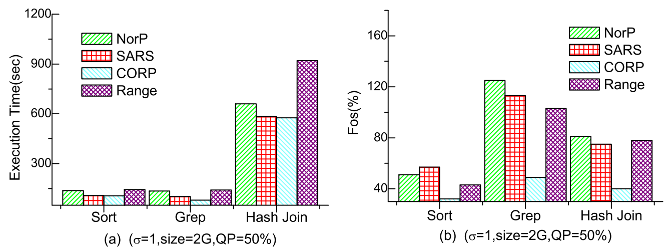


Fig. 12. Whole performance evaluation.

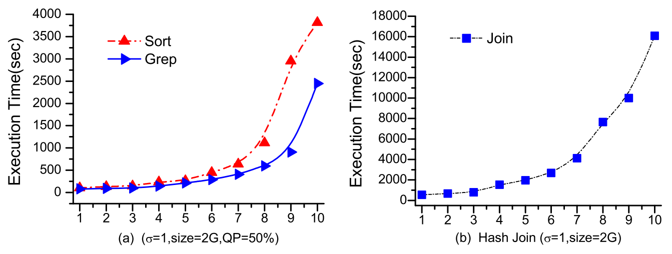


Fig. 13. The performance evaluation for batch jobs.

7 Conclusion

The existence of data skew in the intermediate output created by map tasks provides an opportunity to optimize the cross-rack communication by placing the reduce tasks on the appropriate nodes. This paper mainly involves the following work, i.e., a sampling method based on a reservoir algorithm is applied to sample the data selection. We propose an evaluation model and undertake a great deal of experimental research to select the appropriate sample rate. The advantage of this model is the ability to comprehensively consider the importance of cost, effect, and variance in sampling.

Sampling is an independent MapReduce job, which can output a distribution matrix of intermediate results in each partition. Based on this, by calculating the distance and cost matrices among the cross-node communication, the related map and reduce tasks can be scheduled to relatively nearby physical nodes for data locality. Experiments verify that the inner data transmission can be obviously optimized through this algorithm. For most highly skewed data, the job execution time can also be decreased for lower inner data communication.

Acknowledgments

The work is supported by the National Natural Science Foundation of China (Grant Nos. 61572176), National Hightech R&D Program of China (2015AA015303), the Key Technology Research and Development Programs of Guangdong Province (2015B010108006), and International S&T Cooperation Program of China (2015DFA11240). Zhuo Tang is the corresponding author.

REFERENCES

[1] N. Tiwari, S. Sarkar, and U. Bellur, "Classification framework of MapReduce scheduling algorithms," *ACM Comput. Surveys*, vol. 47, no. 3, pp. 1–38, 2015.

[2] X. Lu, F. Liang, B. Wang, L. Zha, and Z. Xu, "DataMPI: Extending MPI to hadoop-like big data computing," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2014, pp. 829–838.

[3] L. Jiang, P. Patel, G. Ostrouchov, and F. Jamitzky, "OpenMP-style parallelism in data-centered multicore computing with R," *ACM Sigplan Notices*, vol. 47, no. 8, pp. 335–336, 2014.

[4] Hadoop.[eb/ol]. (2016, June). [Online]. Available: http://hadoop.apache.org

[5] 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.

[6] Q. Chen, J. Yao, and Z. Xiao, "LIBRA: Lightweight data skew mitigation in MapReduce," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 9, pp. 2520–2533, Sep. 2015.

[7] J. Tan, S. Meng, X. Meng, and L. Zhang, "Improving reducetask data locality for sequential MapReduce jobs," in *Proc. IEEE INFOCOM*, 2013, pp. 1627–1635.

[8] M. Bourguiba, K. Haddadou, I. El Korbi, and G. Pujolle, "Improving network I/O virtualization for cloud computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 673–681, Mar. 2014.

[9] L.-Y. Ho, J.-J. Wu, and P. Liu, "Optimal algorithms for cross-rack communication optimization in MapReduce framework," in *Proc. IEEE Int. Conf. Cloud Comput.*, 2011, pp. 420–427.

[10] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proc. ACM SIGOPS 22nd Symp. Operating Syst. Principles*, 2009, pp. 261–276.

[11] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "MapReduce with communication overlap (MaRCO)," *J. Parallel Distrib. Comput.*, vol. 73, no. 5, pp. 608–620, 2013.

[12] N. Maheshwari, R. Nanduri, and V. Varma, "Dynamic energy efficient data placement and cluster reconfiguration algorithm for MapReduce framework," *Future Generation Comput. Syst.*, vol. 28, no. 1, pp. 119–127, 2012.

[13] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, and L. Qi, "LEEN: Locality/fairness-aware key partitioning for MapReduce in the cloud," in *Proc. IEEE 2nd Int. Conf. Cloud Comput. Technol. Sci.*, 2010, pp. 17–24.

[14] M. Hammoud and M. F. Sakr, "Locality-aware reduce task scheduling for MapReduce," in *Proc. IEEE 3rd Int. Conf. Cloud Comput. Technol. Sci.*, 2011, pp. 570–576.

[15] J. Tan, S. Meng, X. Meng, and L. Zhang, "Improving reducetask data locality for sequential MapReduce jobs," in *Proc. IEEE INFOCOM*, 2013, pp. 1627–1635.

[16] B. Palanisamy, A. Singh, L. Liu, and B. Jain, "Purlieus: Locality-aware resource allocation for MapReduce in a cloud," in *Proc. Int. Conf. High Performance Comput. Netw. Storage Anal.*, 2011, Art. no. 58.

[17] 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.

[18] K. Wang, X. Zhou, T. Li, D. Zhao, M. Lang, and I. Raicu, "Optimizing load balancing and data-locality with data-aware scheduling," in *Proc. IEEE Int. Conf. Big Data*, 2014, pp. 119–128.

[19] Z. Tang, L. Jiang, J. Zhou, K. Li, and K. Li, "A self-adaptive scheduling algorithm for reduce start time," *Future Generation Comput. Syst.*, vol. 43, pp. 51–60, 2015.

[20] R. P. Mount, et al., "The office of science data-management challenge," Doe Office of Advanced Scientific Computing Research March—May, vol. 6, no. 2, pp. 15–16, 2004.

[21] 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.

[22] E. Ardizzoni, A. A. Bertossi, M. C. Pinotti, S. Ramaprasad, R. Rizzi, and M. V. Shashanka, "Optimal skewed data allocation on multiple channels with flat broadcast per channel," *IEEE Trans. Comput.*, vol. 54, no. 5, pp. 558–572, May 2005.

[23] 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, Dec. 1993.

[24] S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries," *ACM SIGMOD Record*, vol. 29, no. 2, pp. 487–498, 2000.

[25] J. W. Kim, S.-H. Cho, and I. Kim, "Improving efficiency in range-based query processing on the hadoop distributed file system by leveraging partitioned tables," *Information*, vol. 17, no. 10(B), pp. 5311–5316, 2014.

[26] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in MapReduce based on scalable cardinality estimates," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 522–533.

[27] J. Tan, X. Meng, and L. Zhang, "Coupling task progress for MapReduce resource-aware scheduling," in *Proc. IEEE INFOCOM*, 2013, pp. 1618–1626.

[28] J. Lin, et al., "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, vol. 1, pp. 57–62.

[29] Reservoir sampling. [eb/ol]. (2016, May). [Online]. Available: https://en.wikipedia.org/wiki/Reservoir_sampling

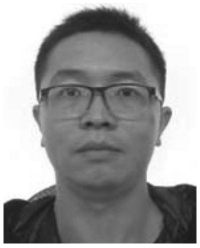
[30] R. Grover and M. J. Carey, "Extending MapReduce for efficient predicate-based sampling," in *Proc. IEEE 28th Int. Conf. Data Eng.*, 2012, pp. 486–497.

[31] F. Yuanquan, W. Weiguo, X. Yunlong, and C. Heng, "Improving MapReduce performance by balancing skewed loads," *China Commun.*, vol. 11, no. 8, pp. 85–108, 2014.

[32] Ratings and classification data. [eb/ol]. (2016, Apr.). [Online]. Avaialable: http://webscope.sandbox.yahoo.com

[33] 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.

Zhuo Tang received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2008. He is currently an associate professor in the College of Computer Science and Electronic Engineering, Hunan University, and an associate chair in the Department of Computing Science. His majors are distributed computing system, cloud computing, and parallel processing for big data, including distributed machine learning, security model, parallel algorithms, and resources scheduling and management in these areas. He is a member of the ACM and the CCF.



Wen Ma received the BASc degree in computer science from the University of Science and Technology Liaoning, China. And now he is working toward the master's degree in the College of Information Science and Engineering, Hunan University, China. His research interests include parallel computing, improvement and optimization of task scheduling module in Hadoop and Spark platforms.

Kenli Li received the PhD degree in computer science from Huazhong University of Science and Technology, China, in 2003. Now he is a professor of computer science and technology with Hunan University, and an associate director of National Supercomputing Center, Changsha. His major research includes parallel computing, grid and cloud computing, and DNA computer. He has published more than 320 journal articles, book chapters, and refereed conference papers. And he is an outstanding member of the CCF and a member of the IEEE.

Keqin Li is a SUNY distinguished professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things, and cyber-physical systems. He has published more than 410 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of the *IEEE Transactions on Parallel and Distributed Systems*, the *IEEE Transactions on Computers*, the *IEEE Transactions on Cloud Computing*, and the *Journal of Parallel and Distributed Computing*. He is a fellow of the IEEE.