Journal of Visual Languages and Computing 46 (2018) 1–11

Contents lists available at ScienceDirect

Journal of Visual Languages and Computing

journal homepage: www.elsevier.com/locate/jvlc

NPIY : A novel partitioner for improving mapreduce performance

Wei Lu[a], Lei Chen[a,b,*], Liqiang Wang[b], Haitao Yuan[a], Weiwei Xing[a], Yong Yang[a]

[a] School of Software Engineering, Beijing Jiaotong University, Beijing, China
[b] Department of Computer Science, University of Central Florida, Orlando, USA

A R T I C L E I N F O

Article history:
Received 2 November 2017
Revised 12 April 2018
Accepted 24 April 2018
Available online 24 April 2018

MSC:
00-01
99-00

Keywords:
MapReduce
Hadoop
Data skew
Load balance
Data transmission amount
Heterogeneous
Parallel image processing

A B S T R A C T

MapReduce is an effective and widely-used framework for processing large datasets in parallel over a cluster of computers. Data skew, cluster heterogeneity, and network traffic are three issues that significantly affect the performance of MapReduce applications. However, the hash-based partitioner in the native Hadoop does not consider these factors. This paper proposes a new partitioner for Yarn (Hadoop 2.6.0), namely, NPIY, which adopts an innovative parallel sampling method to distribute intermediate data. The paper makes the following major contributions: (1) NPIY mitigates data skew in MapReduce applications; (2) NPIY considers the heterogeneity of computing resources to balance the loads among Reducers; (3) NPIY reduces the network traffic in the shuffle phase by trying to retain intermediate data on those nodes running both map and reduce tasks. Compared with the native Hadoop and other popular strategies, NPIY can reduce execution time by up to 41.66% and 58.68% in homogeneous and heterogeneous clusters, respectively. We further customize NPIY for parallel image processing, and the execution time has been improved by 28.8% compared with the native Hadoop.

1. Introduction

The past decade has witnessed a proliferation of data generation and processing techniques [1]. MapReduce [2] has been proven to be an effective tool to process large data sets. As a parallel computing framework that supports MapReduce, Apache Hadoop [3] is widely used in many different fields. such as parallel image processing. MapReduce consists of two main functions: the map function, which transforms input data into intermediate data, namely $<$key,value$>$ pairs, and the reduce function, which summarizes the values with the same key. Partitioning [4] is a critical to MapReduce because it determines the Reducer to which an intermediate data item will be sent in the shuffle phase. Hadoop 2.6.0 employs a static hash function to partition the intermediate data, which is called Hash-Partitioner and shown in Eq. (1). Although MapReduce is currently gaining wide popularity in parallel data processing, its Hash-Partitioner in Hadoop is still not ideal and has room to be improved.

$$Hash(Hashcode(Intermediate\ data)\ mod\ ReducerNum) \tag{1}$$

First, data skew [5] is one of the most critical impact factors affecting the performance of Hadoop cluster. Data skew refers to the imbalance in terms of data allocated to each task or work required to process. When data skew occurs, the aforementioned Hash-Partitioner could lead to a scenario that most of nodes remain idle after they complete their tasks [24]. Thus, this approach prolongs execution time and decreases computing efficiency. Therefore, balancing hash partition size, which is defined as the size of the $<$key,value$>$ pairs with the same hash result, is an important indicator for balancing the loads among Reducers [25].

Secondly, heterogeneity is neglected by the Hash-Partitioner. The computing environments for MapReduce in the real world are typically heterogeneous [6]. Even if intermediate data is not skew, the execution time of tasks in different nodes are diverse because of various computing capacities, consequently, stragglers still exist in clusters [26]. Therefore, Hash-Partitioner does not work well in a heterogeneous Hadoop cluster.

Thirdly, with the increasing size of computing clusters [7], it is common that many nodes run both map tasks and reduce tasks. Obviously, the more intermediate data stay on these nodes, the less network traffic happens in the shuffle phase [8]. However, the Hash-Partitioner does not consider this fact [27]. Therefore, large amount of data transmitted seriously decreases the performance of cluster.

* Corresponding author at: School of Software Engineering, Beijing Jiaotong University, Beijing, China.
E-mail addresses: 13112084@bjtu.edu.cn, my_shitou320@163.com (L. Chen).

https://doi.org/10.1016/j.jvlc.2018.04.001
1045-926X/© 2018 Elsevier Ltd. All rights reserved.

Among the previous solutions, some are specific to a particular type of applications [9,10], some require a pre-sampling of input data [11–13], and some identify the tasks with the greatest expected remaining processing time and repartition unresolved data to fully utilize nodes in cluster [14]. There are lots of studies [15,16] on heterogeneous Hadoop cluster to reduce network traffic in the shuffle phase. In our previous work [17], we proposed a method named PIY to improve the performance of MapReduce by comprehensively considering the three deficiencies mentioned above; however, we find it still has a lot of room to improve.

This paper proposes a novel partitioner for Yarn (Hadoop 2.6.0), namely, NPIY (Novel Partitioner in Yarn), to solve the problems of data skew and network traffic in the shuffle phase in heterogeneous Hadoop clusters. Compared with the previous studies, the contributions of this paper can be summarized as follows:

(1) We propose a novel sampling method, named PRS. PRS achieves a highly accurate approximation to the distribution of intermediate data by sampling the input data during normal map processing, and its overhead is very low and negligible. We also propose an evaluation model to select an appropriate sample percentage in PRS. This model comprehensively considers the importance of cost, effect, and variance in sampling.
(2) To tackle the data skew problem, we propose a strategy called BASHE, which is based on an Approx-Relaxed-Subset-Sum algorithm. BASHE effectively mitigates load imbalance among Reducers.
(3) To avoid the performance degradation caused by heterogeneity, NPIY allocates appropriate amount of intermediate data to Reducers according to their computing capacities.
(4) NPIY optimizes network traffic by decreasing the amount of transmitted data on the nodes executing both map and reduce tasks.

We evaluate the performance of NPIY in YARN (Hadoop 2.6.0). Compared with other popular strategies, NPIY can reduce the execution time by up to 41.66% and 58.68% in homogeneous and heterogeneous Hadoop clusters, respectively. We also evaluate NPIY on a parallel image processing application. Compared with several existing strategies, NPIY can reduce the execution time by up to 11.2%.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3.3 briefly introduces the *Approx-Relaxed-Subset-Sum* algorithm used by NPIY. Section 4 describes our NPIY in details. Section 5 describes the performance evaluation of NPIY. Finally, Section 6 concludes this paper.

2. Related work

To estimate the distribution of intermediate result before determining the partition in Hadoop, sampling methods are widely used in previous studies. We classify these methods into two categories. The first category is to launch a pre-run extra job to conduct data distribution statistics before the whole normal job, and then to decide an appropriate partition [11]. The drawback of these methods is that when data volume is large, sampling will cost much time and delay the start of map tasks, which prolongs the execution time of whole job. The second category is to integrate sampling into the map stage [5]. However, these methods hardly achieve high sampling accuracy, and also cause performance degradation because the parallel degree is decreased between the map and reduce stages.

Data skew has also been studied in MapReduce environment in the past few years. Ibrahim et al. [18] propose LEEN, which partitions all intermediate keys according to their frequencies and the fairness of the expected data distribution after the shuffle phase. However, LEEN lacks preprocessing to estimate data distribution effectively. Gufler et al. propose TopCluster [19] to mitigate data skew among Reducers by estimating the cost of each intermediate partition. However, it increases the intermediate data transmission amount in the shuffle phase because it ignores data locality in the reduce phase.

Heterogeneous computing environment is a research hot spot in recent years. LATE [15] calculates the progress rate of tasks and reassigns the task with the longest remaining time to other idle nodes. The work in [6] presents a system that adopts virtualization technology to allocate data center resources dynamically based on the application's demands. A set of heuristics is developed to combine different types of workloads so as to reduce overload effectively. However, these approaches cannot solve the data skew problem well.

In addition, all the aforementioned approaches ignore the fact that there are heterogeneous nodes running both map tasks and reduce tasks concurrently in large-scale computing cluster. The network traffic in the shuffle phase will be optimized remarkably if the partitioner is able to dramatically reduce the transmission amount of intermediate data among those nodes. Our approach, *i.e.*, NPIY, can comprehensively resolve these problems mentioned above.

3. Approx-Relaxed-Subset-Sum algorithm

In our previous work [17], we propose an approach called PIY to reduce our problem into the well-know Subset-Sum problem. In a heterogeneous cluster, the computing capacities of different Reducers are diverse. Let T denote the computing capacity of a Reducer. The problem of load balance can be modeled as a *Subset-Sum Problem*, which is abbreviated as *SS* in this paper. An instance of the *SS* problem is a pair (K, T), where K is a set of n positive integers (in arbitrary order), *i.e.*, $k_1, k_2, \ldots, k_n$, and T is a positive integer. A *SS* problem is to find whether there exists a subset of K that add up as large as possible but not greater than the target value T, which is a NP-complete problem. The load balance problem can be approximately solved by calling multiple rounds of *SS*, each of which is to determine data assignment for a given Reducer i with a capacity T_i, except for the tail Reducer, which may have more leftovers than its capacity. Such a tail Reducer may cause load imbalance.

Example. Assume there are 3 Reducers in a cluster, the intermediate data set contains 6 $<$key,value$>$ pairs, and the amount of pairs are denoted by K={ k_0=105, k_1=115, k_2=123, k_3=138, k_4=145, k_5=151}. For simplicity, we assume that the three Reducers are homogeneous, which means that they have equal computing capacity with a target value T=258. When using the Approx-Subset-Sum algorithm introduced in our previous work [17] (in order to clarify the problem more clearly, in the Approx-Subset-Sum algorithm, we eliminate the Trim function, which is shown as Algorithm 2 in Section 3.3), the final amount of intermediate data partitioned to 3 Reducers are 256 ($k_0 + k_5$), 253 ($k_1 + k_3$), 268 ($k_2 + k_4$) and their standard deviation is 6.48. Obviously, this result is not good because the last Reducer is assigned with much more intermediate data than the other two.

The result can be explained as the *SS* problem is constrained by the restriction that the sum of selected integers must be less than or equal to the target value. As far as selecting intermediate data is concerned, the total amount of selected intermediate data must be less than or equal to T. Therefore, in the shuffle phase, Reducers whose intermediate data are assigned firstly usually have total load T, and these missing loads are taken over by the other Reducers whose intermediate data are assigned later. This decreases the degree of load balance.

3.1. Relaxed-Subset-Sum problem

To solve this issue, we propose a more accurate approach, namely, **Relaxed-Subset-Sum problem**, which can be formally stated as: given n positive integers, $k_0, k_2, \dots, k_{n-1}$, and target value T, we try to find a subset of these integers whose sum is *as close to T as possible*. We call this problem *RSS* for short. The *RSS* problem can be formulated as the following integer program:

$$min|t-T| \quad s.t. \quad t=\sum_{i=1}^{n} y_i k_i, \quad y_i \in \{0,1\}, i=1,2,\dots n$$

y_i =1 indicates k_i is selected in the subset, and y_i=0 otherwise.

Both *RSS* and *SS* problem are defined *w.r.t* a set *K* of *n* positive integers k_i $(0<i<n-1)$ and target value *T*, which are denoted by *RSS(K, T)* and *SS(K, T)*, respectively. That is, $y_1^*, y_2^*, \dots, y_s^*$ is an optimal solution of *RSS* problem,

$$RSS(K,T)=\sum_{i=1}^{n} k_i y_i^*$$

Similarly, we denote the optimal sum by *SS(K, T)*.

In order to facilitate the discussion of algorithm in Section 3.2, we give two properties of the *RSS* problem and their proofs. The first property describes the relationship between the *RSS* problem and *SS* problem.

Property 1. *Given a set K that consists of* n *positive integers* $k_0, k_1, \dots, k_{n-1}$ *and a target value* T, *if* $y_0^*, y_1^*, \dots, y_{n-1}^*$ *is an optimal solution for a RSS problem and* RSS(K, T) $\le$ T, *then* $y_0^*, y_1^*, \dots, y_{n-1}^*$ *is also an optimal solution for an SS problem, i.e.,* SS(K, T)=RSS(K, T).

Proof. Suppose $y_0^*, y_1^*, \dots, y_{n-1}^*$ is not an optimal solution for this *SS* instance, there must be another solution $y'_0, y'_1, \dots, y'_{n-1}$ to this *SS* instance, such that

$$\sum_{i=0}^{n-1} k_i y_i^* < \sum_{i=0}^{n-1} k_i y'_i \le T$$

So we have $|\sum_{i=0}^{n-1} k_i y_i^* - T| > |\sum_{i=0}^{n-1} k_i y'_i - T|$. That means the sum obtained from $y'_0, y'_1, \dots, y'_{n-1}$ is closer to T than that of $y_0^*, y_1^*, \dots, y_{n-1}^*$, contradicting the assumption that $y_0^*, y_1^*, \dots, y_{n-1}^*$ is an optimal solution of the *RSS* instance. So $y_0^*, y_1^*, \dots, y_{n-1}^*$ must be an optimal solution, and *RSS(K, T)=SS(K, T)*=$\sum_{i=0}^{n-1} k_i y_i^*$. □

The second property shows that the difference between the values of *RSS(K,T)* and *T* cannot be too much.

Property 2. *Given a set K consisting ofnpositive integersk*$_0, k_1, \dots, k_{n-1}$*and a target value*T, *a*RSS*problem has an optimal solution*$y_0^*, y_1^*, \dots, y_{n-1}^*$, *then* $RSS(K,T)-k_j<T$ *for any* $y_j^*=1 (0\le j\le n-1)$.

Proof. This property is obviously true when *RSS(K, T)* $\le$ *T*, so we only focus on the case of *RSS(T)* $>$ *T*.

Let $RSS_j(K,T)$ denote $RSS(K,T)-k_j$ for any y_j^*=1. It is assumed that $RSS_j(K,T)$ $>$ *T*, and we have $|RSS_j(K,T)-T| = |RSS(K,T)-k_j-T| = RSS(K,T)-k_i-T < RSS(K,T)-T = |RSS(K,T)-T|$. According to the definition of *RSS* problem, RSS_j is the optimal solution. However, this contradicts the definition of *RSS(K, T)*. Therefore, we must have $RSS_j<T$, which is equivalent to $RSS(K,T)-k_j<T$. □

In summary, when *RSS(K, T)* $\le$ *T*, Property 1 guarantees the *RSS* problem is equivalent to the *SS* problem. When *RSS(K, T)* $>$ *T*, Property 2 guarantees that the difference between *RSS(K, T)* and *T* must be less than the largest k_i $(0<i<n-1)$. In practice, if the amount of intermediate data received by a Reducer is beyond its computing capacity, Property 2 ensures that the degree of the beyond is not too much and will not cause much overhead. This has been verified in our experiments. Having demonstrated these properties, we are ready to explore algorithms for the *RSS* problem.

Algorithm 1 Relaxed-Subset-Sum Algorithm.

```
Input: K: a positive integer set containing $n$ elements $<k_0,\dots,k_{n-1}>$;L: a positive integer set;T: a target value;$L_i$: a generated list after $k_i$ is appended.
Output: an optimal solution
1: $n$ = the length of K
2: $L_0$ = 0
3: for i = 1 to n do
4:     $L_i$ = MERGE-LIST($L_{i-1}, L_{i-1}+k_i$)
5:     $L_i$ = PRESERVE($L_i$, $T$)
6: end for
7: t1, t2 = LARGEST-TWO($L_n$)
8: if $|t_i-T|<|t_2-T|$ then
9:     $t^*=t_1$
10: else
11:     $t^*=t_2$
12: end if
13: Back trace from $t^*$ to get an optimal solution.
```

Algorithm 2 Trim Algorithm.

```
Input: L: a positive integer set contains $m$ factors $<l_0,\dots,l_{m-1}>$;$\varepsilon$: trimming parameter;
Output: $L'$
1: $m$ = L.length;
2: $L'=\langle l_0\rangle$;
3: last = $l_0$;
4: for i = 1 to m-1 do
5:     if $l_i > last*(1+\varepsilon)$ then
6:         append $i_1$ onto the end of $L'$;
7:         last = $l_i$;
8:     end if
9: end for
10: return $L'$;
```

3.2. Relaxed-Subset-Sum algorithm

In this section, we propose our Relax-Subset-Sum algorithm, which is shown in Algorithm 1. The algorithm's inputs are operation loads and target value, the algorithm generates n+1 integer sets $L_i (0\le i\le n)$. Line 2 initializes the list L_0 to 0. The loop in lines 3 - 7 computes L_i as a sorted list containing a properly trimmed version of L_{i-1}. MERGE-LISTS(L, L') in line 4 returns a sorted list that is the merge of its two sorted input lists L and L' with duplicate values removed. Specifically, L_{i-1} + k_i denotes the list of integers generated by adding k_i to each element of L_{i-1}. For example, if L_{i-1} = ⟨1, 2, 3, 5, 9⟩, L_i = L_{i-1} + 2 = ⟨3, 4, 5, 7, 11⟩. Function PRESERVE(L_i, T) in line 5 perseveres all the elements smaller than T, and the smallest one among those elements is larger than or equal to T. When the loop terminates, L_i contains the sum of subsets of $\{k_1, k_2, \dots, k_n\}$.

Function LARGEST-TWO(L_n) in line 7 selects the maximum two integers in L_n, which are denoted by t_1 and t_2. Lines 8–12 select the one closer to the target value T from t_1 and t_2. The selected element is denoted by t^*. Obviously, t^* is an optimal sum. In line 13, we back trace integer sets $L_0, L_1, \dots, L_n$ to obtain the solution.

Example. We illustrate the Relaxed-Subset-Sum algorithm with the same example mentioned before. The final amount of intermediate data assigned to the three Reducers are 260 (k_1+k_4), 256 (k_2+k_3) and 261 (k_0+k_5), and their standard deviation is only

Algorithm 3 Approx-Relaxed-Subset-Sum Algorithm.

```
Input: K: a positive integer set contains n elements $< k_0, \ldots, k_{n-1} >$;L: a positive integer set;T: a target value;$L_i$: a generated list after the $k_i$ is appended.
Output: an optimal solution
1: $n$ = the length of K
2: $L_0$ = 0
3: for i = 1 to n do
4:     $L_i$ = MERGE-LIST($L_{i-1}, L_{i-1} + K_i$)
5:     $L_i$ = Trim($L_{i-1}$, $\varepsilon$)
6:     $L_i$ = PRESERVE($L_i$, $T$)
7: end for
8: t1, t2 = LARGEST-TWO($L_n$)
9: if $\mid t_i - T \mid < \mid t_2 - T \mid$ then
10:    $t^* = t_1$
11: else
12:    $t^* = t_2$
13: end if
14: Back trace from $t^*$ to get an optimal solution.
```

2.16, much smaller than 6.48. Hence, the *RSS* algorithm provides a better load balance than the *SS* algorithm.

We analyze the time complexity of the *RSS* algorithm as follows. Line 1 takes O(1) time. The time of the loop in lines 3–6 depends on the length of L_i. L_i only contains the distinct nonnegative integers smaller than T and at most one integer equal to or larger than T, thus the maximum length of L_i is T $(0 \le i \le n)$. Therefore, the time complexity of this loop is O(nT). Line 7 and the *if* statement in lines 8–12 take O(1) time. The back trace in the last line takes n-1 steps, t^* in L_i determines the time of each step. This can be done through binary search in $O(log|L_i|)$=O(log$T$). Therefore, the last line takes $O(nlogT)$ time, and the total time complexity of *RSS* algorithm is $O(nT)$.

3.3. Approx-Relaxed-Subset-Sum Algorithm

The *RSS* algorithm is time-consuming when the data being processed is extremely large. To reduce the time complexity, we propose an Approx-Relaxed-Subset-Sum algorithm, which implements the *Trim* algorithm. As shown in Algorithm 2, the *Trim* algorithm lists L by selecting and remaining only a value Z to represent all values Y according to Eq. (2) and finally get the list $L^{'}$. Here ε $(0 < \varepsilon < 1)$ is a trimming parameter. Obviously, the *Trim* algorithm can dramatically decrease the number of elements by keeping a close (and slightly smaller) representative value in the list for the deleted elements. Algorithm 2 describes the procedure of trimming list L that contains m elements in time $\Theta(m)$. It is assumed that L is sorted in a monotonically increasing order. The output of the procedure is a trimmed and sorted list.

$$\frac{Y}{1+\varepsilon} \le Z \le Y \tag{2}$$

We insert the algorithm *Trim* into Relaxed-Subset-Sum algorithm and place it just before the function PRESERVE(L_i, T) to form our Approx-Relaxed-Subset-Sum algorithm, which is shown as Algorithm 3. Approx-Relaxed-Subset-Sum algorithm can find a sub-optimal solution in a fully polynomial time, therefore, the caused overhead is acceptable [17].

4. NPIY: A Novel Partitioner in yarn

4.1. System overview

To tackle the problem of system performance degradation caused by data skew, heterogeneity of cluster, and large amount of network traffic, we design a new partitioner in Yarn (Hadoop 2.6.0), named NPIY, Fig. 1 shows its architecture. In particular, each Parallel Reservoir Sampler (PRS) samples the input data on each Mapper. Data Frequency Table (DFT) creates a table that records the value of each key in every DataNode according to sampling statistics. The Capacity Monitor estimates the computing capacity of each DataNode. Global DFT (GDFT) summarizes all DFT data in each DataNode. CV records the computing capacity values of all DataNodes. BASHE is the core unit in our NPIY, which generates the final partitioning result. The workflow of NPIY consists of three steps.

(1) When a user submits a MapReduce job, the Resource Manager in the NameNode creates an Application Master for this job in a DataNode. Besides normal MapReduce tasks, this Application Master is also in charge of Capacity Monitor tasks, PRS tasks, and DFT tasks. Then the Application Master applies for containers to run all these tasks in parallel, which are shown as MapReduce Container, Capacity Monitor Container, PRS Container, and DFT Container in Fig. 1. When the split operation in the map stage finishes, PRS tasks in each DataNode conduct sampling. All the sampled $<$key,value$>$ pairs are summarized and stored into a file. The detailed process of PRS is described in Section 4.2. A DFT task counts and records the sampled $<$key, value$>$ pairs, and generates a key frequency table. Here the key frequency refers to the number of pairs with a specific key. At the same time, the Capacity Monitor tasks collect the computing capacity of each DataNode, which is described in Section 4.3. When all these processes are completed, the obtained information, which consists of key frequencies in DFTs and computing capacity value of each DataNode, will be transmitted from DataNodes to NameNode through heartbeat messages. In Hadoop, DataNodes send heartbeat messages to NameNode after a regular interval to indicate that they are alive and working. In these heartbeat messages, DataNodes also send their computing and resource usage information to NameNode.

(2) When NameNode receives the information, it forwards them to GDFT and CV. Then GDFT summarizes all key frequencies in each DataNode into a total frequency table. CV records computing capacity values of all DataNodes by the information from Capacity Monitors.
After achieving the capacity value of each DataNode, NPIY selects the top 40% of DataNodes with the highest capacity values to process reduce tasks, which ensures the execution parallelism and efficiency of reduce tasks. On these DataNodes, the user-specified number of Reducers are set up with Mappers and reserved during map tasks are executed.
Compared with the native Hadoop, the reserved Reducers do not decrease the utilization of system resources because of the following reasons. First, in the native Hadoop, when a certain small percentage of map tasks complete (default by 5%), Reducers start to pull their input data by using the Hash Partitioner. Therefore, Reducers start to work shortly after Mappers. Secondly, the selected DataNodes are with stronger processing capacity, so the map tasks on them are executed quickly. These are essential preparative works for the final partitioning results generated by BASHE.

(3) Then BASHE generates the final partitioning result using the BASHE algorithm described in Section 4.5. Finally, NameNode transmits the results back to DataNodes through resource response messages.

In NPIY, we obtain the distribution of intermediate result by running a novel sampling during normal map processing. Our sampling is performed in parallel by the map

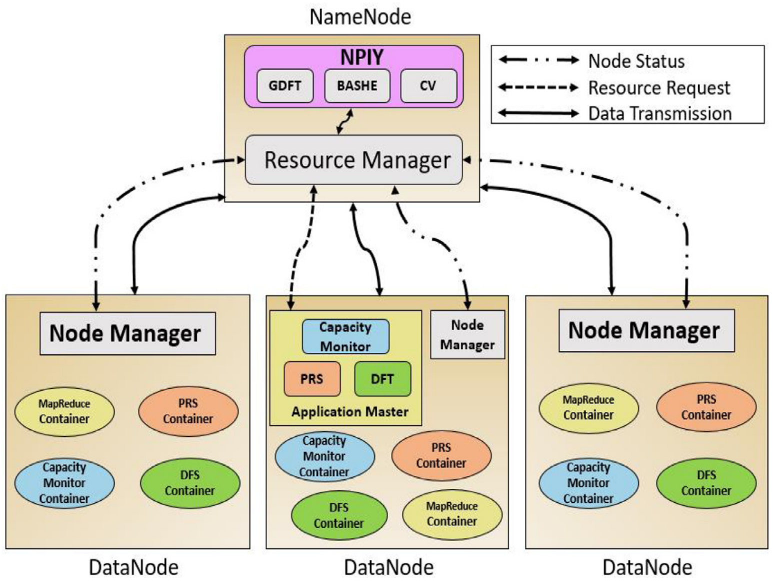


Fig. 1. The Architecture of NPIY.

tasks with higher priority. Therefore, when the split operation in the map stage finishes, the map tasks conducting sampling are processed preferentially. Our sampling strategy, namely Parallel Reservoir Sampling (PRS for short), is based on the reservoir sampling algorithm [20]. PRS runs by invoking the class `org.apache.hadoop.mapreduce.lib.InputSampler` and overloading the `SplitSampler` method. Obviously, there is a tradeoff between sampling overhead and result accuracy. If the sampling rate is too large, a higher sampling accuracy is achieved with the cost of longer execution time. On the contrary, small sampling rate shorts sampling duration but sacrifices sampling accuracy. In our previous work [17], we set sampling rate empirically. In this paper, we propose a more accurate evaluation model to select an appropriate sampling rate, which comprehensively considers the importance of effect, cost, and variance in sampling.

4.2. Parallel reservoir sampling strategy

4.2.1. Sampling strategy

The main idea of PRS is described as follows. PRS builds a reservoir for each selected split and samples k elements from it. All $<$key, value$>$ pairs in each split are scanned and the first k elements are stored in each reservoir. For a $<$key, value$>$ pair whose sequence number is larger than k, we replace the stored elements with this pair based on a certain probability.

This process is executed for each reservoir in parallel. All the sampled $<$key, value$>$ pairs are summarized together and stored into a file by a reduce function. The process of PRS is shown in Fig. 2(a). When sampling tasks of selected splits are finished, their corresponding normal user-defined map functions are executed successively. In other words, our PRS is integrated into a normal map processing. As shown in Fig. 2(b), when all samplers are complete, the sampling results are aggregated and transmitted to the GDFT model in our NPIY, which performs the BASHE algorithm to generate the final partitioning scheme. To those $<$key, value$>$ pairs that do not appear in the PRS results, BASHE assigns them to the corresponding Reducers according to their hash code, which is described in Algorithm 4. In our system, Reducers begin to pull their input data after the sampling partition is decided. This is later than the default start time of the reduce stage in the native Hadoop because the decision of sampling partition introduces

Algorithm 4 BASHE Algorithm.

```
Require: k, r, key_size[0,..,k-1], CV[0,... ,r-1], $Sum\_CV$, $\varepsilon$, RS[0,..,r-1], T[0,... ,r-1], Total_Size
Ensure: key_dest[0,... ,k-1]
1:  for i = 0 to k-1 do
2:      $key\_dest_i = -1$;
3:  end for
4:  for j = 0 to r-1 do
5:      $RS_j = 0$;
6:      $Sum\_CV = Sum\_CV + CV_j$;
7:  end for
8:  for each Reducer $R_j (0 \le j \le r-1)$ do
9:      if the node with $R_j$ locates also executes map tasks then
10:         for every $key_i$ (0 ≤ i ≤ k-1)on $R_j$ do
11:             if $key\_dest_i$ == -1 then
12:                 $key\_dest_i$ = $MaxReducer(key_i)$;
13:                 $RS_{key\_dest_i} = RS_{key\_dest_i} + key\_size_i$;
14:                 $key\_size_i$ = 0;
15:             end if
16:         end for
17:     end if
18: end for
19: for j = 0 to r-1 do
20:     $T_j = Total\_Size * (CV_j / Sum\_CV) - RS_j$;
21: end for
22: for j=0 to r-1 do
23:     $Z^*$=ARSS($key\_size[0, \ldots, k-1], T_j, \varepsilon$);
24:     Set $key\_dest$ of the keys composing $Z^*$ to the sequence number of $Reducer_j$;
25: end for
26: for i = 0 to k-1 do
27:     if $key\_dest_i == -1$ then
28:         $key\_dest_i$ = Hash(Hashcode($Key_i$) mod (the sequence number of Reducer));
29:     end if
30: end for
31: return $key\_dest[0, \ldots, k-1]$;
```

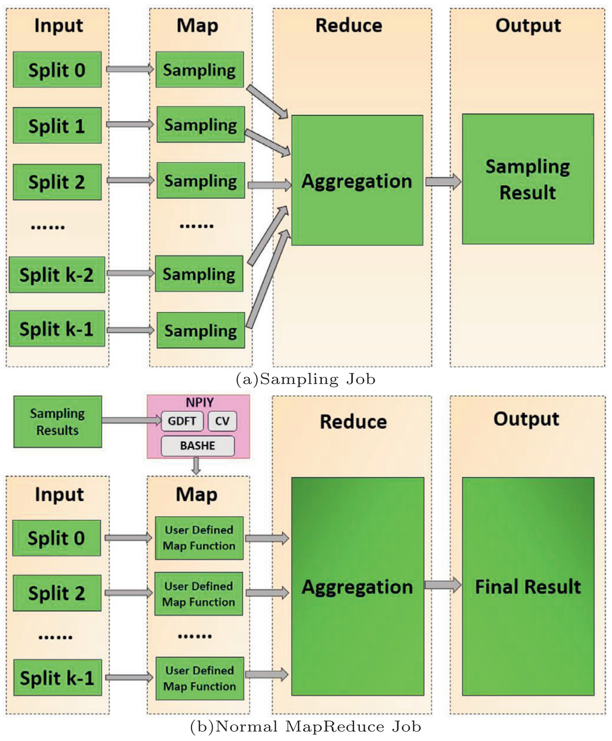


Fig. 2. The Process of Parallel Reservoir Sampling.

overhead. However, the overhead is negligible based on our experimental results shown in Section 5.

4.2.2. Sampling rate selection

Through extensive experiments, we find that the sampling rate affects not only sampling effect, but also sampling duration and variation of these duration. We propose an evaluation model to better estimate appropriate sampling rates and take these affected factors into consideration comprehensively. The evaluation model is shown in Eq. (3),

$$i = argMin[f_i(E_i, D_i, V_i) = \alpha E_i + \beta D_i + \gamma V_i] \tag{3}$$

where function $f_i(E_i, D_i, V_i)$ comprehensively considers both sampling effect, time cost, and variation. α, β, and γ are the weight coefficients that reflect the importance of these affected factors.

In function f_i, E_i denotes the sampling effect when sampling rate is set to i. In this paper, we focus on 5 different sampling percentages: 1%, 25%, 50%, 75%, 100%, so the series number i satisfies $1 \le i \le SN$, where SN denotes the number of different sampling rates, here, SN=5. As shown in Eq. (4), E_i represents sampling effect by the differences among the sequences of CoV (Coefficient of Variation) values between the currently adopted percentage and the whole input dataset.

$$E_i = \sqrt{\sum_{j=1}^{N}\left(cov_{i,j} - \frac{1}{N}\sum_{j'=1}^{N} cov_{5,j'}\right)^2} \tag{4}$$

where $cov_{i,j}$ denotes the CoV value of j^{th} sampling experiment under sampling rate i. N is the repeated times of experiment. For example, $cov_{5,j}$ denotes the values with a 100% sampling rate.

D_i denotes an average sampling duration, which is calculated by Eq. (5).

$$D_i = \frac{1}{N}\sum_{j=1}^{N} d_{i,j} \tag{5}$$

Table 1
Evaluation for cost and effect with different benchmarks.

i	Sampling percentage (%)	E_i	D_i	V_i	f_i
1	1	357.31	285	43.35	685.66
2	25	150.5	401	29.82	581.32
3	50	101.01	504.16	24.68	629.85
4	75	53.11	594.08	19.92	667.11
5	100	12.58	632.82	2.66	648.06

where $d_{i,j}$ represents the execution time of the j^{th} sampling experiment under the i^{th} sampling rate, $1 \le i \le SN$ and $1 \le j \le N$.

Due to the fact that our cluster is heterogeneous, variation of computation time exists in our parallel sampling. To fully consider this factor, we propose Eq. (6) to calculate the variation, which is denoted by V_i under sampling rate i.

$$V_i = \sqrt{\frac{1}{N}\sum_{j=1}^{N}\left(COV_{i,j} - \frac{1}{N}\sum_{j=1}^{N} COV_{i,j}\right)^2} \tag{6}$$

We run *Sort* and *Grep* benchmarks in our experiments. Table 1 presents the final values of E_i, D_i and V_i, which are calculated using Eqs. (4)–(6) with different sampling rate i, respectively. We perform each group of experiments for 10 times, so the parameter N in Eqs. (4)–(6) is set to 10. Here, we simply assume the importance of efficiency, cost and variation are equal, and set $\alpha = \beta = \gamma = 1$.

From Table 1, we observe that with the increase of sampling rate, the value of D_i increases, and both the values of E_i and V_i decrease. Specifically, in the group of experiments with 1% sampling percentage, E_i's value is great larger than the one with 100% sampling percentage, which means although low sampling rate reduces the sampling duration, it can not demonstrate the distribution of input data accurately. According to Eq. (3), it is easy to observe that sampling 25% of the map tasks is an appropriate choice for the input data of *Sort* and *Grep* benchmarks.

4.3. Capacity monitor

Hadoop cluster is often heterogeneous. To obtain DataNodes' computing capacity, we design a model, namely, Capacity Monitor, which runs on each DataNode. To dramatically decrease extra overhead caused from heterogeneity, Capacity Monitor in each DataNode keeps monitoring the sampled input data when sampling begins to run, and gets its consuming volume $Volume(con)_{id}$ $(1 \le id \le m)$ during a period of time Δt. Here m denotes the number of DataNodes. Then we calculate the capacity value of the id^{th} DataNode, i.e., CV_{id}, by the following Eq. (7). Capacity Monitor tasks send the capacity values of all DataNodes to NPIY in NameNode through heartbeat messages.

$$CV_{id} = Volume(cons)_{id}/\Delta t \tag{7}$$

4.4. Network traffic in shuffle phrase

As bandwidth is a scarce resource, the shuffle phase has become the bottleneck of MapReduce due to its large amount of network traffic. Many DataNodes run both map and reduce tasks [12]. If we can keep as many as intermediate <key,value> pairs on these DataNodes by using an appropriate partitioning method in the shuffle phase, it can further decrease the network traffic. The BASHE algorithm finds the DataNode that contains the maximum amount of these pairs, and then transmits all these pairs to this DataNode. Our experimental results show that this method decreases the network traffic in the shuffle phase by up to 22.22%.

Table 2
Variable notation.

Variable name	Description
k	The number of distinct keys
r	The number of Reducers in cluster
key_dest_i	The ID of the destination Reducer that will process the $<$key,value$>$ pairs corresponding to key_i
RS_i	The data volume that should be processed on $Reducer_i$
CV_i	Computing capacity of $Reducer_i$
Sum_CV	The total computing capacity value of all Reducers
key_size_i	The data volume of ⟨key,value⟩ pair with key_i
T_i	Remaining capacity of $Reducer_i$
$Total\_Size$	The total volume of experimental data set
ε	Approximation parameter

4.5. BASHE Algorithm

In this section, we describe our algorithm BASHE that comprehensively considers the load **BA**lance among Reducers, network traffic in **S**huffle and the **H**eterogeneity of Hadoop cluster basing on Approx-R**E**laxed-Subset-Sum algorithm. As Algorithm 4 shows, there are three steps in BASHE. First, it reduces intermediate data transmitted in the shuffle phase. Then, it predicts the data volume that each Reducer should process according to their computing capacity. Finally, to balance the loads among Reducers, BASHE partitions intermediate data to each Reducer using the Approx-Relaxed-Subset-Sum algorithm. The variables used in BASHE are shown in Table 2.

Lines 1–3 initialize all key_dest_i with -1, which means all $<$key,value$>$ pairs have not been partitioned. Lines 4–7 initialize all RS_i $(1 < i < r)$, and compute the Sum_CV. The value of CV_i can be obtained by Eq. (7). Lines 8–18 reduce the amount of network traffic in the shuffle phase. As described in Section 4.4, we focus on the DataNodes who run both map and reduce tasks. For each key_i $(0 < i < k-1)$ on these Reducers, BASHE first checks whether its destination Reducer is determined. If not, the function MaxReducer(key_i) in line 12 will find the Reducer on which the volume of the $<$key,value$>$ pairs with key_i is maximum, and then set this Reducer as the destination Reducer of key_i. Line 13 updates data volume that should be processed on this Reducer. Line 14 sets key_size_i to zero, because the $<$key,value$>$ pairs with key_i have been partitioned to a Reducer, this operation prevents these $<$key,value$>$ pairs from being re-partitioned. Lines 19–21 obtain the remaining computing capacity of each Reducer. Here *remaining computing capacity* means extra data volume that one Reducer can process. Total_Size * (CV_j/Sum_CV) means the total data volume that the $Reducer_j$ should process according to its computing capacity.

Using the Approx-Relaxed-Subset-Sum algorithm (abbreviated as ARSS in line 23), Lines 22–30 balance the loads among all Reducers by partitioning intermediate data based on Reducers' computing capacity. In Lines 22–25, BASHE partitions intermediate data to each Reducer and records the destination Reducer of each key into the array key_dest. We can obtain these values through GDFT in NPIY. In lines 26–30, BASHE checks whether there exist the keys that have not been partitioned, *e.g.*, the keys not appeared in PRS results. If exists, BASHE assigns these intermediate data to the corresponding Reducers according to their hash code. The last line returns partitioning result.

5. Evaluation

In this section, we describe the performance evaluation of NPIY by running two popular benchmarks with synthetic and real-world datasets with different data skew rate, and our experiments are performed under both homogeneous and heterogeneous environments. Specifically, we evaluate NPIY to process large-sized images in parallel.

Table 3
Jobs with different sampling methods.

Sampling Method	Time(s)	Sample File Size(MB)	$Accu_{appro}$
Random	5.7	2.8	310,986
TopCluster	5.1	3.0	147,282
PRS	5.3	10.6	99,763

5.1. Experimental environment

In our experiments, we set up two Hadoop clusters, one is homogeneous, and the other is heterogeneous. Our homogeneous Hadoop cluster consists of 60 physical machines installed with Ubuntu 12.04 (KVM as the hypervisor) with 16 core 2.53 GHz Intel processors, 4G memory, and the 60 nodes are connected through a single switch with 1 Gbps network bandwidth. Our experiments are performed in YARN (Hadoop 2.6.0). All nodes are used as both computing and storage nodes. The HDFS block size is set to 64 MB and each node is configured to run at most 6 map tasks and 2 reduce tasks concurrently. Our heterogeneous cluster contains 60 physical machines with three types. The first type contains 30 machines with 16 core 2.53GHz Intel processor and 2 GB memory. The second type contains 20 machines with 4 core 3.3 GHz Intel processor, and 8 GB memory. The third type contains 10 machines with 4 core 2.4 GHz Intel processor, and 2 GB memory. The other configurations are the same as that in the homogeneous cluster.

We evaluate NPIY by running different types of benchmarks in homogeneous and heterogeneous Hadoop clusters, respectively. We compare NPIY with Hadoop hash partition, SkewTune [14], and our prior approach PIY [17]. In order to ensure accuracy, we perform each group of experiments at least 10 times and take the mean value as the final result.

5.2. Accuracy of the sampling method

To evaluate how our sampling method PRS can achieve a good approximation to the distribution of intermediate result, we run a *Sort* benchmark and compare our PRS with the other two sampling methods. One is the random sampler used in the native Hadoop and the other is TopCluster [19]. We run these three different samplers on a 20GB real-word data sets from the full English Wikipedia archive, which contains 50,000 keys. Our sampling experiments are executed on our homogeneous Hadoop cluster, each node is configured to run at most 6 map tasks and 2 reduce tasks concurrently. We measure the sampling approximation by Eq. (8), where x_i^{appro} and x_i^{real} denote the sampling and real frequency of ⟨key, value⟩ pairs with key_i, respectively. The smaller value $Appro_{sampl}$ is, the better performance our approach achieves. As described in Section 4.2.2, we set sample percentage to 25%. In order to maintain consistency, the other two methods sample 25% of input splits. Table 3 shows that the size of sample file generated from PRS is larger than the others meanwhile their execution time are approximately equal. This is because PRS completes reservoir sampling on each split in parallel and collects the sample result with larger volume. Obviously, the better accuracy can be realized if the sampling result is larger.

$$Appro_{sampl} = \sqrt{\frac{\sum_{i=1}^{n}(x_i^{appro} - x_i^{real})^2}{n}} \tag{8}$$

From Table 3 we can see that the approximation of our PRS is 99,763, which is better than the other two sampling methods.

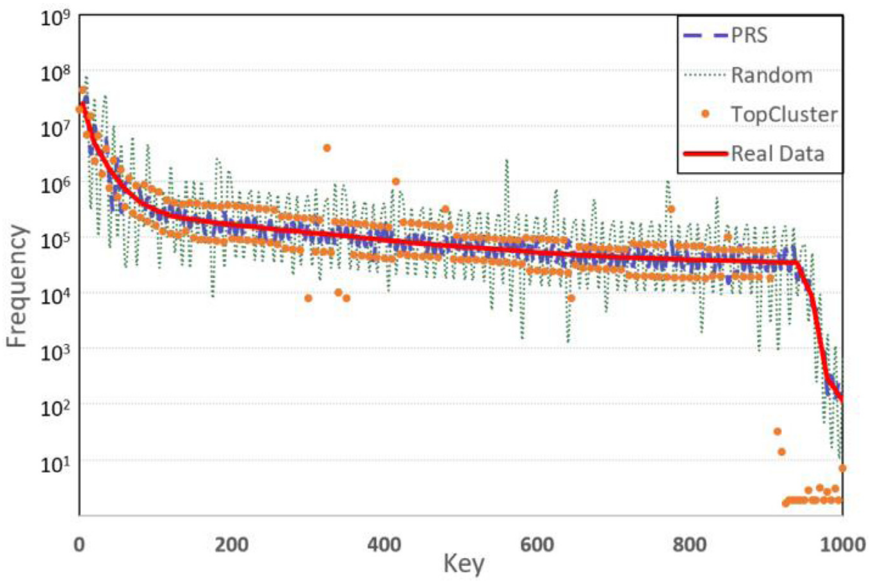


Fig. 3. Comparison of three sampling methods in Grep.

Fig. 3 for the top 1000 large keys in the data shows the same result. Note that the sampling approximation of TopCluster is fairly accurate on the large keys at the beginning of its curve, representing their frequency are relatively large, but it becomes worse on the keys whose frequency is less than 10^3. The reason is that TopCluster assumes the distribution of small keys are in accordance with large keys, and this assumption can be misleading when there are a large number of small keys in the input data. Fig. 3 shows that our PRS achieves a better approximation to the distribution of intermediate data in MapReduce computation.

5.3. Load balance among reducers when running sort benchmark

One of major motivations for NPIY is to balance the loads among Reducers when data skew happens. Therefore, we evaluate NPIY by running the *Sort* benchmark, which is a reduce-input-heavy job to process input data with various data skew degrees. In this paper, the degree of load balance and data skew are measured by the coefficient of variation, which is represented as COV. The smaller COV means the better. Fig. 4(a) and (b) shows the COV values when running *Sort* benchmark based on 10 GB of synthetic data in homogeneous and heterogeneous clusters, respectively. We generate a 10GB synthetic data set following Zipf [21] distributions with varying δ parameters from 0.2 to 1.2 to control the degree of the skew.

We compare our NPIY with three strategies, Hadoop-Hash, SkewTune [14], and PIY [17]. As shown in Fig. 4(a), the curves of Hadoop-Hash and SkewTune keep rising when data skew rate increases, while the COV of NPIY always remains very low. This can be explained as NPIY partitions the intermediate data to all Reducers evenly in the homogeneous cluster. The reason why SkewTune performs worse than NPIY and PIY is that SkewTune can only repartition the input data of one straggler at a time, it can not balance the loads on all Reducers when there are more than one slow reduce tasks caused by serious skew data. On account of the Hadoop-Hash partitions data by the hash code of keys, it is easy to cause serious unbalanced loads when data skew happens, which makes the worst performance in Fig. 4(a).

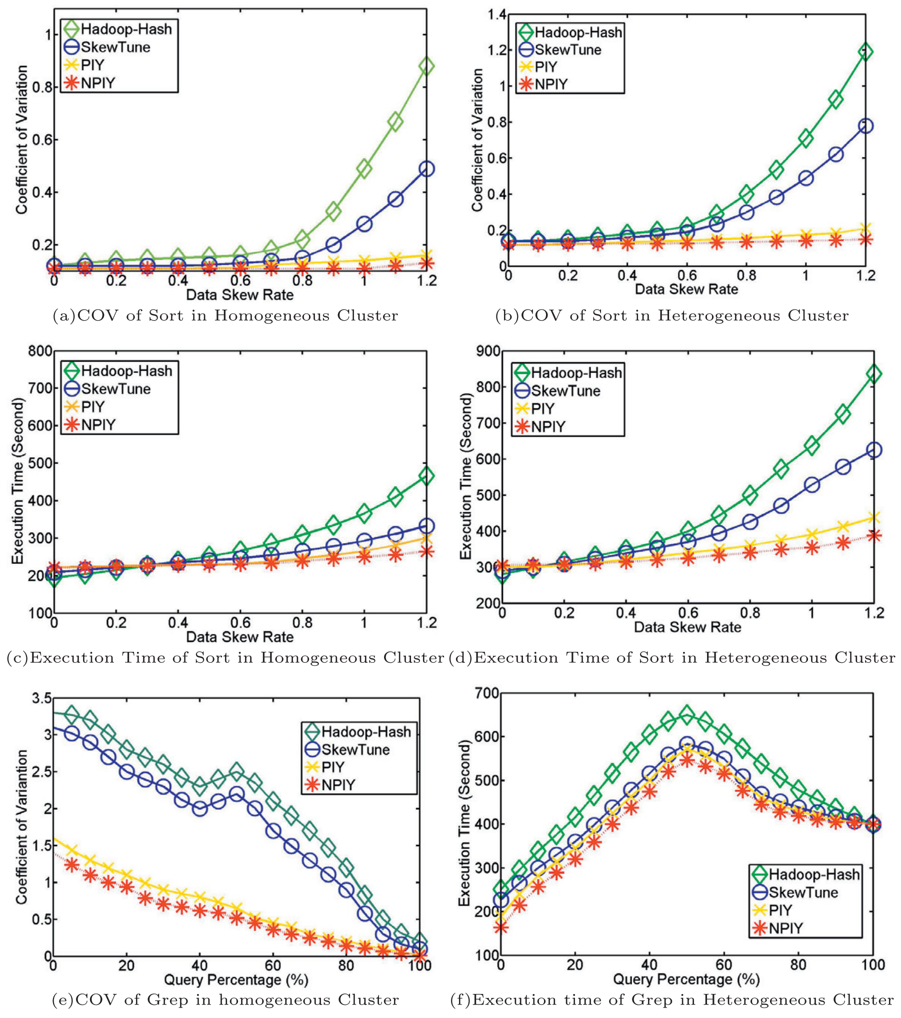
From Fig. 4(b), we find the same results as in Fig. 4(a). NPIY is also with the best performance in terms of COV, and the Hadoop-Hash is the worst one. In addition, while the value of NPIY is almost unchanged, the values of Hadoop-Hash and SkewTune at most of data skew rates are higher than that in the homogeneous cluster, and this trend is more obvious when the data skew rate increases. In other words, the optimization degree of NPIY in load balance in the heterogeneous cluster is much more than that in the homogeneous cluster. Besides the aforementioned reasons, the consideration of heterogeneity of NPIY makes a greater contribution in load balance among Reducers.

From Figs. 4(a) and (b), we find that, compared with PIY, NPIY does better in all time. This can be explained as the Approx-Subset-Sum algorithm in PIY causes that the last Reducers are allocated with more intermediate data than the front ones, which decreases the degree of load balance. Hence, the Approx-Relaxed-Subset-Sum algorithm in NPIY effectively avoids the occurrence of this situation.

5.4. Execution time of sort benchmark

Fig. 4 also shows the execution time of the experiments we have described in Section 5.3. The curves in Fig. 4(c) show the results in the homogeneous cluster. We can see that NPIY is faster than Hadoop-Hash and SkewTune when processing the data with high skew rate. On the contrary, when the data skew rate is lower than a certain threshold, NPIY does not perform satisfactorily. The reason is that when data skew degree is low, *e.g.* less than 0.30 in our experiment, the Hadoop-Hash has the shortest execution time in the homogeneous cluster because of its even partitions of intermediate data without extra overhead. SkewTune causes small overhead on migrating unprocessed data on the slower nodes to the faster ones because there are few stragglers in this scenario, and this leads to its execution performance behinds the Hadoop-Hash. Furthermore, NPIY consumes the longest execution time because of its extra overhead produced by the sampling data and partitioning results in the map phase. However, as data skew degree increases, the optimization, which is achieved through balancing loads among Reducers by using our Approx-Relaxed-Subset-Sum algorithm, gradually offsets the time spent on the extra overhead. Therefore, NPIY achieves the shortest execution time.

In addition, NPIY is always faster than PIY because it does better in load balance, which avoids stragglers produced by PIY and shortens the execution time. In addition, as described in Section 3.3, although the Approx-Relaxed-Subset-Sum algorithm may cause loads on some Reducers slightly beyond their comput-



(a)COV of Sort in Homogeneous Cluster (b)COV of Sort in Heterogeneous Cluster

(c)Execution Time of Sort in Homogeneous Cluster (d)Execution Time of Sort in Heterogeneous Cluster

(e)COV of Grep in homogeneous Cluster (f)Execution time of Grep in Heterogeneous Cluster

Fig. 4. Evaluation of PIY In Hadoop Clusters.

ing capacity, the optimization on load balance offsets this overhead.

Consequently, compared with Hadoop-Hash, SkewTune and PIY, NPIY achieves the average improvements on execution time by 20.39%, 8.60% and 4.65%, respectively, and the maximum improvements reach 41.66%, 14.90% and 7.81%, respectively, when the data skew percentage is 1.2.

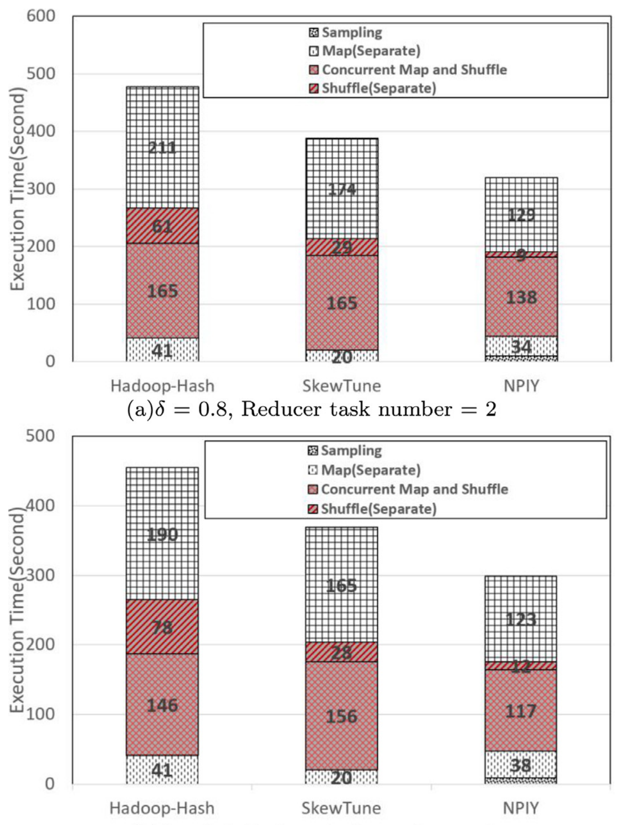
Fig. 4(d) shows the full adaption of NPIY to heterogeneous cluster. NPIY is also the fastest one in most cases. There also exists a skew rate threshold in the heterogeneous cluster, under which the PIY and NPIY run slightly slower than the other two methods, the reason is the same as in the homogeneous cluster. However, on average, NPIY can perform 31.86% and 16.65% faster than Hadoop-Hash and SkewTune, respectively. Specifically, when the data skew is 1.2, the improvement is 58.68% and 30.92%, respectively. These values demonstrate that the improvement degree of NPIY is more obvious in the heterogeneous cluster than that in the homogeneous cluster because it considers the computing capacity of each node during partitioning. For the same reason we have explained before, compared with PIY, NPIY performs 5.38% faster on average and 8.89% when data skew is 1.2.

5.5. Grep benchmark testing

To evaluate the performance of NPIY on reduce-input-light applications, we run Grep, a reduce-light job, in our heterogeneous cluster. We modify 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 use is the full English Wikipedia archive with a total data size of 10 GB. 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%. Figs. 4(e) and (f) shows that NPIY obtains the best performance of COV and job execution time at all time due to the accuracy of PRS and the consideration of heterogeneity. Specifically, in Fig. 4(e), NPIY obtains the best COV when the query percentage is lower. This is because PRS in NPIY is good at searching unpopular words in the archive and generates better sampling results. As the query percentage increases, the distribution of the result data becomes increasingly uniform, so the performance improvement vanishes. From these two figures, we also observe that, compared with SkewTune, the optimization NPIY achieves is not so much. This can be explained as Grep is a reduce-input-light application, thus the amount of intermediate data is relativity small, and the room to improve is not much. With the same reason we have described, the performance of NPIY in load balance and duration are all better than PIY.

5.6. Optimization in shuffle phase

To verify that BASHE algorithm used by NPIY can decrease the amount of data transmission in the shuffle phase, we record the



(a)$\delta = 0.8$, Reducer task number = 2

(b)$\delta = 0.8$, Reduce task number = 4

Fig. 5. The Execution Time Of Each Phase.

execution time in each phase of MapReduce when running *Sort* job in the heterogeneous cluster. Without losing generality, we illustrate the execution time when δ is 0.8 in Fig. 4(d). The native Hadoop starts shuffle tasks when 5% map tasks finish, therefore, we divide MapReduce into 4 phases, which are represented as Map (Separate), Concurrent Map and Shuffle, Shuffle (Separate), and Reduce. Concurrent Map and Shuffle indicate the overlap period in which the shuffle tasks begin to run and map tasks have not totally finished. Therefore, the duration of Map phase equals the sum of Map (Separate) and "Concurrent Map and Shuffle". Similarly, the duration of Shuffle phase, which is shown in red color in Fig. 5, equals to the sum of Shuffle(Separate) and "Concurrent Map and Shuffle". In particular, because NPIY executes PRS in the Map phase, its duration of Map phase should contain additional time costed by PRS, which is represented as Sampling in Fig. 5. From Fig. 5(a), we can see the duration in the shuffle phase of NPIY is 138+9=147 seconds, which is less than SkewTune (165+29=194) and Hadoop-Hash (165+61=226), the improvement are $\frac{194-147}{194} =$ 24.23% and 34.96%, respectively.

Through a plenty of experiments, we observe that, compared with Hadoop-Hash and SkewTune, the improvement degree NPIY achieves in the shuffle phase is in proportion to the number of reducers until the degree reaches the peak value. In our experiment, the peak improvement degree is achieved when each node is configured to run 6 map tasks and 4 reduce tasks concurrently. Compared with the original configuration (6 map tasks and 2 reduce tasks), this modification should increase the number of Reducers because the native Hadoop determines Reducer nodes according to the computing resource (container in Yarn) in each Reducer. The results are shown in Fig. 5(b). We can easily observe that NPIY's duration in the shuffle phase is 117+12=129 seconds, which is much less than 284 seconds for SkewTune, and 224 seconds for Hadoop-Hash, the improvement is up to $\frac{284-129}{284} = 54.58\%$ and 42.41%, which are larger than Fig. 5(a). This can be explained as with the number of Reducers increases, the BASHE algorithm finds much input data whose map and reduce tasks are able to be scheduled to the same DataNode. This results in the decrease of amount of data transmission in the shuffle phase. However, when we configure each node to run at most 6 map tasks and 6 reduce tasks concurrently, compared with SkewTune and Hadoop-Hash, the improvement caused by NPIY are reduced to 22.53% and 29.31%, respectively. How to find the optimal Reducer number is a problem about the tradeoff between the computing parallelism degree and the network traffic, which is our future work on NPIY.

Table 4
The size of sample images.

Image name	Size(in bytes)
CARTOSAT-1	1,342,552,576
CARTOSAT-2A	4,259,355,002
CARTOSAT-2B	9,204,661,322

5.7. NPIY In parallel image processing

With the demand of processing large-sized images increases rapidly, parallel image processing technologies are widely used to shorten the execution time. MapReduce has become a suitable platform for large-scale high-volume image processing applications because of the following three reasons: (1) images can be easily represented as multiple dimensional structure; (2) majority of functions in image processing can be highly parallelized; (3) the HDFS is an efficient way to store image data.

NPIY is applicable to large-sized image processing due to the following reasons. First, the RGB and gray values of pixels in an image are always unevenly distributed, *i.e.*, these values are skew. There are many popular image processing algorithms that do calculation with these values, such as auto-contrast, histogram, Sobel filtering, sharpening. For example, auto-contrast algorithm filters out a certain percentage of higher values and lower values in Red, Green and Blue channels. Therefore, the value skew decreases the efficiency of these algorithms dramatically. Secondly, the computing clusters processing large-sized image are always heterogeneous in real production environments.

We conducted experiments for images with large sizes approximately from 1.3 GB to 9.1 GB Chinese Remote Sensing (IRS) satellite series. The sample data sets are shown in Table 4. We conduct histogram [22] operation on the native Hadoop, HIPI [23], PIY, and NPIY. HIPI is an open-source Hadoop Image Processing Interface, which processes image without requiring the additional coding by using Java Image Processing Library. Histogram operation counts the frequency of the pixel intensity in an entire image, which is similar to counting words in a file. In our implementation, the map function splits the large-sized image into several pieces. One piece is processed by one map task, which collects the count of the pixel (gray) value. Reduce function completes aggregation of the collected numbers from map functions. To increase amount of input data in the reduce phase, we add TeraSort operation in histogram and finally output pixel intensity results in a descending order. We implement histogram operation in a 5-node heterogeneous cluster, which is composed of the three types of physical computers described in Section 5.1. Specifically, a node in the first type acts as a master, two nodes for each of the other types act as slaves.

As shown in Fig. 6, NPIY obtains the shortest execution time when processing all 3 different large-sized images. Compared with the other three methods, the average execution time is reduced by 28.8%, 19.2%, and 8.2%, respectively due to the following rea-

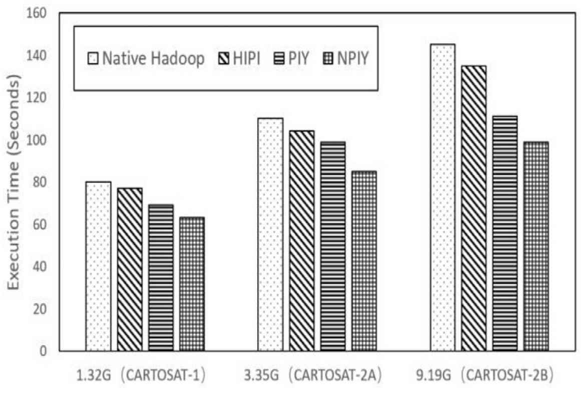


Fig. 6. Execution Time Of Histogram.

sons. First, the distribution of the frequency of the pixel intensity in a large-sized image is not even in general, *i.e.*, the pixel intensity values are skew. Different from the native Hadoop and HIPI, NPIY considers data skew by balancing the loads on Reducers. Secondly, the PRS sampling method helps NPIY realize more even distribution of pixel intensity than the other two frameworks. Thirdly, heterogeneity consideration helps NPIY achieve the fastest process speed. Fourthly, the better load balance makes NPIY run faster than PIY.

6. Conclusion

This paper proposes NPIY to mitigate data skew in MapReduce systems. Using the parallel reservoir sampling method we proposed, NPIY achieves even distribution of intermediate data with negligible overhead. NPIY tries to reduce network traffic in the shuffle phase by decreasing data traffic on those nodes running both map and reduce tasks. NPIY also considers cluster heterogeneity when balancing loads among Reducers. We run Sort and Grep benchmarks on two 10 GB synthetic and real-world data sets, respectively. Compared with some other popular strategies, NPIY improves performance by 41.66% and 58.68% in the homogeneous and heterogeneous clusters, respectively. In particular, in a parallel imagine processing application, NPIY can reduce the execution time by up to 28.8%.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China (Nos. 61272353,61428201,1622292),China Postdoctoral Science Foundation(2016M 600912), Program for New Century Excellent Talents in University (NCET-13-0659), Beijing Higher Education Young Elite Teacher Project(YETP0583).

Supplementary material

Supplementary material associated with this article can be found, in the online version, at doi:10.1016/j.jvlc.2018.04.001.

References

[1] H. Yuan, J. Bi, W. Tan, M. Zhou, B. Li, J. Li, Ttsa: an effective scheduling approach for delay bounded tasks in hybrid clouds, IEEE Trans. Cybern. 47 (11) (2017) 3658–3668.
[2] L. Chen, W. Lu, X. Che, W. Xing, L. Wang, Y. Yang, Mrsim: mitigating reducer skew in mapreduce, in: Proceedings of the 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), IEEE, 2017, pp. 379–384.
[3] S. Konstantin, K. Hairong, R. Sanjay, C. Robert, The hadoop distributed file system, in: Proceedings of the 26th IEEE Symposium on Mass storage Systems and Technologies (MSST), IEEE, 2010, pp. 1–10.
[4] H. Zhang, H. Huang, L. Wang, MRapid: an efficient short job optimizer on hadoop, in: Proceedings of the the 31st IEEE International Parallel Distributed Processing Symposium (IPDPS), IEEE, 2017.
[5] Q. Chen, J. Yao, Z. Xiao, Libra: lightweight data skew mitigation in mapreduce, IEEE Trans. Parallel Distrib. Syst. 26 (9) (2015) 2520–2533.
[6] X. Zhen, S. Weijia, C. Qi, Dynamic resource allocation using virtual machines for cloud computing environment, IEEE Trans. Parallel Distrib. Syst. 24 (6) (2013) 1107–1117.
[7] H. Yuan, J. Bi, W. Tan, B. Li, Temporal task scheduling with constrained service delay for profit maximization in hybrid clouds, IEEE Trans. Autom. Sci. Eng. 14 (1) (2017) 337–348.
[8] H. Zhang, L. Wang, H. Huang, Smarth: Enabling multi-pipeline data transfer in hdfs, in: Proceedings of the 43rd International Conference on Parallel Processing (ICPP), IEEE, 2014, pp. 30–39.
[9] K. YongChul, B. Magdalena, H. Bill, R. Jerome, Skew-resistant parallel processing of feature-extracting scientific user-defined functions, in: Proceedings of the 1st ACM Symposium on Cloud computing, ACM, 2010, pp. 75–86.
[10] V. Subramanian, L. Wang, E.-J. Lee, P. Chen, Rapid processing of synthetic seismograms using windows azure cloud, in: Proceedings of the IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2010, pp. 193–200.
[11] X. Yujie, Q. Wenyu, L. Zhiyang, L. Zhaobin, J. Changqing, L. Yuanyuan, L. Haifeng, Balancing reducer workload for skewed data using sampling-based partitioning, Comput. Electr. Eng. 40 (2) (2014) 675–687.
[12] V. Subramanian, H. Ma, L. Wang, E.-J. Lee, P. Chen, Rapid 3d seismic source inversion using windows azure and amazon ec2, in: Proceedings of the 2011 IEEE World Congress on Services, in: SERVICES '11, IEEE, 2011, pp. 602–606.
[13] H. Huang, L. Wang, E.-J. Lee, P. Chen, An mpi-cuda implementation and optimization for parallel sparse equations and least squares (lsqr), Proc. Comput Sci 9 (2012) 76–85.
[14] K. YongChul, B. Magdalena, H. Bill, R. Jerome, Skewtune: mitigating skew in mapreduce applications, in: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, ACM, 2012, pp. 25–36.
[15] Z. Matei, K. Andy, J. Anthony, K. Randy, S. Ion, Improving mapreduce performance in heterogeneous environments., in: Proceedings of the Osdi, 8, 2008, p. 7.
[16] L. Jun, L. Feng, A. Nirwan, Monitoring and analyzing big traffic data of a large-scale cellular network with hadoop, IEEE Netw. 28 (4) (2014) 32–39.
[17] W. Lu, L. Chen, H. Yuan, W. Xing, L. Wang, Y. Yang, Improving mapreduce performance by using a new partitioner in yarn, in: Proceedings of the 23rd International Conference on Distributed Multimedia Systems, Visual Languages and Sentient Systems, IEEE, 2017, pp. 24–33.
[18] I. Shadi, J. Hai, L. Lu, W. Song, H. Bingsheng, Q. Li, Leen: locality/fairness-aware key partitioning for mapreduce in the cloud, in: Proceedings of the IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2010, pp. 17–24.
[19] G. Benjamin, A. Nikolaus, R. Angelika, K. Alfons, Load balancing in mapreduce based on scalable cardinality estimates, in: Proceedings of the IEEE 28th International Conference on Data Engineering (ICDE), IEEE, 2012, pp. 522–533.
[20] V. Jeffrey, Random sampling with a reservoir, ACM Trans. Math. Softw. (TOMS) 11 (1) (1985) 37–57.
[21] A. Lada, H. Bernardo, Zipfâ;;s law and the internet, Glottometrics 3 (1) (2002) 143–150.
[22] K.J. Narain, S. Prasanna, W. Andrew, A new method for gray-level picture thresholding using the entropy of the histogram, Comput. Vision Gr. Image Process. 29 (3) (1985) 273–285.
[23] S. Chris, L. Liu, A. Sean, L. Jason, Hipi: a Hadoop Image Processing Interface for Image-based Mapreduce Tasks, Chris. University of Virginia, 2011.
[24] H. Huang, L. Wang, B.C. Tak, L. Wang, C. Tang, Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances, in: Proceedings of the IEEE Sixth International Conference on Cloud Computing (CLOUD), IEEE, 2013, pp. 228–235.
[25] S. Diersen, E.-J. Lee, D. Spears, P. Chen, L. Wang, Classification of seismic windows using artificial neural networks, Proc. Comput. Sci. 4 (2011) 1572–1581.
[26] H. Huang, J.M. Dennis, L. Wang, P. Chen, A scalable parallel lsqr algorithm for solving large-scale linear system for tomographic problems: a case study in seismic tomography, Proc. Comput. Sci. 18 (2013) 581–590.
[27] P. Guo, H. Huang, Q. Chen, L. Wang, E.-J. Lee, P. Chen, A model-driven partitioning and auto-tuning integrated framework for sparse matrix-vector multiplication on gpus, in: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery, ACM, 2011, p. 2.