Improving MapReduce Performance by Using a New Partitioner in YARN

Wei Lu[1,], Lei Chen[1,*], Haitao Yuan[1], Weiwei Xing[1], Liqiang Wang[2], Yong Yang[1]
[1] School of Software Engineering,Beijing Jiaotong University, Beijing, China
[2] Department of Computer Science, University of Central Florida, Orlando, USA
Email: [1]{luwei,13112084,htyuan,wwxing,12112088}@bjtu.edu.cn
[2]{lwang}@cs.ucf.edu

Abstract

Data skew, cluster heterogeneity, and network traffic are three issues that significantly influence the performance of MapReduce applications. However, the Hash-Partitioner in native Hadoop does not consider them. This paper proposes a new partitioner in Yarn (Hadoop 2.6.0), namely, PIY, which adopts an innovative parallel sampling method to achieve the distribution of the intermediate data. Based on this, firstly, PIY mitigates data skew in MapReduce applications. Secondly, PIY considers the heterogeneity of the computing resource to balance the load among Reducers. Thirdly, PIY reduces the network traffic in shuffle phase by trying to retain intermediate data on those nodes who act as both mapper and reducer. Compared with the native Hadoop and some other popular strategies, PIY can reduce the execution time by 35.62% *and* 50.65% *in homogeneous and heterogeneous cluster, respectively. We also implement PIY in parallel image processing. Compared with several existing strategies, PIY can reduce the execution time by* 11.2%

MapReduce; Hadoop; data skew; load balance; data transmission amount; heterogeneous

1 Introduction

MapReduce has been proven to be an effective tool to process large data sets [10]. As a parallel computing framework that supports MapReduce, Apache Hadoop [6] is widely used in many different fields. 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 is applied to list of values that correspond to the same key. Partitioning[4] is a critical feature of MapReduce because it determines the reducer to which an intermediate data item will be sent in shuffle phase. Hadoop 2.6.0 usually employ static hash functions to partition the intermediate data, which is called Hash-Partitioner and described as the formula (1). Although MapReduce is currently gaining wide popularity in parallel data processing, its Hash-Partitioner is still inefficient and has room for improvement.

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

First, data skew [2] is one of the most serious impact factors affecting the performance of Hadoop cluster. Data skew refers to the imbalance in terms of data allocated to each task or the imbalance in terms of work required to process such data. When data skew occurs, the aforementioned Hash-Partitioner leads to the fact that most of nodes have to remain idle after they complete their tasks and await the stragglers. Finally this approach prolongs the execution time and decreases the computing efficiency. Therefore, balancing the hash partition size, which is defined as the size of the key-value pairs with the same hash result, is an important indicator for load balancing among the reducers.

Secondly, heterogeneity is neglected by the Hash-Partitioner. The computing environments for MapReduce in the real world always are heterogeneous [17]. Even if data skew does not happens on intermediate data, the execution time of each node are diverse because their various computing capacities, consequently, stragglers still exist in clusters. Therefore, Hash-Partitioner can not work well in heterogeneous Hadoop cluster.

Thirdly, with the increasing size of computing clusters, it is common that many nodes act as both Mapper and Reducer in real production environment. Obviously, the more intermediate data stay in these nodes, the less network traffic happen in shuffle phase[20]. However, the Hash-Partitioner does not consider this fact.

Many studies have focused on the data skew mitigation. Among the proposed solutions, some are specific to a particular type of applications [9][13], some require a pre-sample of the input data [18][12][16], some identify the task with the greatest expected remaining processing time and repar-

DOI reference number: 10.18293/DMSVLSS2017-002

titions the unresolved data in a way that fully utilizes the nodes in the cluster [10]. There are lot of studies on the heterogeneous Hadoop cluster [19] to reduce network traffic in shuffle phase [11]. However, all previous studies can not well solve the three deficiencies mentioned above comprehensively.

This paper proposes a new partitioner for Yarn (Hadoop 2.6.0), namely, PIY, to solve the problems about data skew and network traffic in shuffle phase in heterogeneous Hadoop cluster. 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 the intermediate data by parallelly sampling the input data during the normal map processing, and it only causes little overhead.
(2) We propose an algorithm, namely BASH, to tackle the data skew problem.
(3) To avoid the degradation performance caused by heterogeneity, PIY allocates appropriate amount of intermediate data to reducers according to their computing capacity.
(4) PIY optimizes the network traffic by decreasing the amount of the transmitted data located on nodes acting as both Mapper and Reducers.
(5) We conduct a performance evaluation with PIY in YARN (Hadoop 2.6.0). Compared with some other popular strategies, PIY can reduce the execution time by 35.62% and 50.65% in homogeneous and heterogeneous Hadoop cluster, respectively. We also implement PIY in parallel image processing. Compared with several existing strategies, PIY can reduce the execution time by 11.2%

The rest of this paper is organized as follows. Section 2 reviews related studies. Section 3 briefly introduces the $Approx\_Subset\_Sum$ algorithm, which is used by PIY. Section 3 describes our PIY in detail. Section 5 describes the performance evaluation of PIY. Finally, Section 6 concludes this paper.

2 Related Work

To ascertain the distribution of the intermediate result before determining the partition in Hadoop, sampling methods are widely applied in previous studies. We classify these methods into two categories. The first category is to launch a pre-run extra job before whole normal jobs to conduct data distribution statistics, and then decide an appropriate partition [18]. The drawback of these methods is that when the volume of data is large, sampling will cost much time which results in prolonging the execution time of whole job. The second category contains the methods that integrate sampling into the map stage [2]. However, these methods hardly achieve high sampling accuracy, and also cause performance degradation because the parallel degree is decreased between the map and the reduce stage.

Data skew has also been studied in the MapReduce environment during the past few years. In [6], Ibrahim et al. proposed 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 the data distribution effectively and separates map and reduce tasks absolutely, and therefore, it incurs significant time cost. Gufler et al. proposed TopCluster [3], which can mitigate data skew among reducers by estimating the cost of each intermediate partition. However, it increases the intermediate data transmission amount in shuffle because it ignores data locality in reduce side.

Heterogeneous computing environment is a research hotspot in recent years. LATE [19] calculates the progress rate of tasks and selects the slow task with the longest remaining time to back up. The work in [17] presents a system that adopts the virtualization technology to allocate data center resources dynamically based on application demands. Their common limitation is that they cannot solve the data skew problem.

However, all the aforementioned approaches ignore the fact that there are plenty of nodes that run map tasks and Reduce tasks concurrently in large-scale computing cluster. The network traffic in shuffle phase will be optimized obviously if the partitioner can reduce the transmission amount of the intermediate data that stay on those nodes. Our approach, i.e., PIY, can comprehensively resolve all problems mentioned in this section.

3 Approx-Subset-Sum Algorithm

Because our PIY algorithm is based on the Approx-Subset-Sum algorithm[8], we introduce it in this section. An instance of the subset-sum problem is a pair (L, t), where *L* is a set $x_1, x_2, ..., x_n$ of *n* positive integers (in arbitrary order) and *t* is a positive integer. This decision problem is to find whether there exists a subset of *L* that adds up exactly to the target value *t*. As we known, this problem is NP-complete and traversing all subset of *L* will take exponential time, and therefore, this is unacceptable when the data being processed is extremely large. To reduce the time complexity, the Approx-Subset-Sum algorithm trims list *L* by selecting and remaining only one value *Z* to represent all the values *Y* according to the formula (2) and finally get the list $L^{'}$. Here ε $(0 < \varepsilon < 1)$ is a trimming parameter. We assume there is a *Z* that represents *y* in the new list $L^{'}$. Each removed element *y* is represented by a remaining element z that satisfies formula (2). Obviously, trimming can dramatically decrease the number of elements kept while remaining a close (and slightly smaller) representative value in the list for each deleted element. Algorithm 1 describes the pro-

cedure of trimming list L that contains m elements in time $\Theta(m)$. It is assumed that L is sorted in monotonically increasing order. The output of the procedure is a trimmed and sorted list.

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

Algorithm 1 Trim Algorithm

Input: **L**: a positive integer set contains m factors $< l_0, ..., l_{m-1} >$; ε: trimming parameter;
Output: $L^{'}$

```
1: m = L.length;
2: L' = ⟨l_0⟩;
3: last = l_0;
4: for i = 1 to m-1 do
5:     if l_i > last * (1 + ε) then
6:         append i_1 onto the end of L';
7:         last = l_i;
8:     end if
9: end for
10: return L';
```

Algorithm 2 Approx-Subset-Sum

Input: **S**: a positive integer set contains n elements $< S_0, ..., S_{n-1} >$; **L**: a positive integer set; **t**: target value; ε(**0**$<\varepsilon<$**1**): trimming parameter; L_i: the generated list after the S_i is appended.
Output: z^*: the largest value in L_n

```
1: n = the length of L
2: L_0 = < 0 >
3: for i=0 to n-1 do
4:     L_i = Merge-Lists(L_{i-1}, L_{i-1} + S_i)
5:     L_i = Trim(L_i, ε/2n)
6:     remove every element that is greater than t from L_i.
7: end for
8: return z*
```

The Approx-Subset-Sum algorithm is described as Algorithm 2. It returns a value z^* whose value is within a 1+ε factor of the optimal solution. Line 2 initializes the list L_0 to be the list containing just the element 0. For loop in lines 3 - 6 computes L_i as a sorted list containing a suitably trimmed version of the set L_{i-1} , with all elements larger than t removed. MERGE-LISTS(L, $L^{'}$) in line 4 returns the sorted list that is the merge of its two sorted input lists L and $L^{'}$ with duplicate values removed. L_{i-1} + S_i denotes the list of integers derived from L_{i-1} by increasing each element of L_{i-1} by S_i. For example, if L_{i-1} = $\langle 1, 2, 3, 5, 9 \rangle$, then $L_i = L_{i-1} + 2 = \langle 3, 4, 5, 7, 11 \rangle$. Trim($L_i$, $\varepsilon/2$n) in line 5 decreases the length of L_i with the trimming parameter $\varepsilon/2$n.

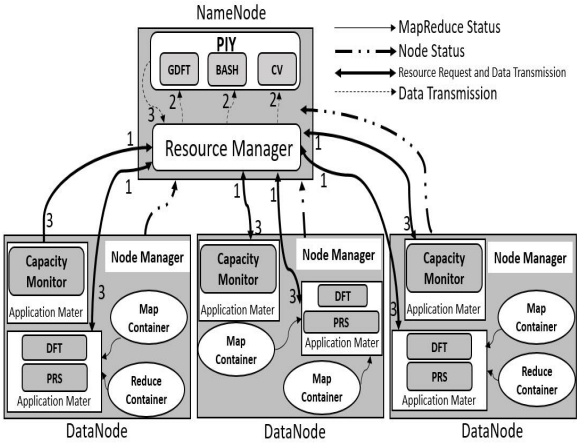


Figure 1: The Architecture of PIY

Here is an example to illustrate the execution of Approx-Subset-Sum algorithm. It is assumed that the instance S=$\langle 104, 102, 201, 101 \rangle$, t=308 and ε=0.40. The trimming parameter is $\varepsilon/8 = 0.05$. $Approx\_Subset\_Sum$ computes the following values in the indicated lines:
line 2: $L_0 = \langle 0 \rangle$
line 4: $L_1 = \langle 0, 104 \rangle$
line 5: $L_1 = \langle 0, 104 \rangle$
line 6: $L_1 = \langle 0, 104 \rangle$

line 4: $L_2 = \langle 0, 102, 104, 206 \rangle$
line 5: $L_2 = \langle 0, 102, 206 \rangle$
line 6: $L_2 = \langle 0, 102, 206 \rangle$

line 4: $L_3 = \langle 0, 102, 201, 206, 303, 407 \rangle$
line 5: $L_3 = \langle 0, 102, 201, 303, 407 \rangle$
line 6: $L_3 = \langle 0, 102, 201, 303 \rangle$

line 4: $L_4 = \langle 0, 101, 102, 201, 203, 302, 303, 404 \rangle$
line 5: $L_4 = \langle 0, 101, 201, 302, 404 \rangle$
line 6: $L_4 = \langle 0, 101, 201, 302 \rangle$

The algorithm returns z^* = 302 as its answer, which is maintaining within 2% of the optimal answer 307 = 104 + 102 + 101. This shows that the $Approx\_Subset\_Sum$ algorithm can find an approx optimal solution in a fully polynomial time, therefore, the overhead it causes is acceptable.

4 A New Partitioner in Yarn

4.1 System Overview

We designed a new partitioner in Yarn, named PIY, based on Hadoop 2.6.0, and the architecture of PIY frame is shown in Figure 1. In particular, each Parallel Reservoir Sampler (PRS) samples the input data on each Mapper.

The Data Frequency Table (DFT) creates a table that records the value of each key in each DataNode according to the sampling statistics. The Capacity Monitor fetches the computing capacity value of each DataNode. The Global DFT (GDFT) summarizes all DFT data in each DataNode. The CV records the computing capacity values of all DataNodes. The BASH is the core unit in our PIY that generates the final partitioning result. The workflow of PIY consists of 3 steps:

(1) When split operation in map stage finishes, PRS applies to Resource Manager for containers to conduct sampling. All the sampled <key,value> pairs are summarized and stored into a file. The detailed process of PRS is described in the following Section 4.2. The DFT counts and records the sampled <key,value> pairs in each DataNode and generates a key frequency table. Here the key frequency refers to the number of pairs corresponding to each key. At the same time, the Capacity Monitor collects the computing capacity of each DataNode, which is described in Section 4.3. When all these processes are completed, the information data, which consists of the key frequencies in DFT and the computing capacity value of each DataNode will be transmitted from the Application Master to the Resource Manager through heartbeat messages.

(2) When the Resource Manager receives the information data, it will transmit corresponding data to the GDFT and CV, respectively. Then the GDFT summarizes all the key frequencies in each DataNode into a total frequency table. The CV records the computing capacity values of all DataNodes by the data from the Capacity Monitor. These are essential preparative works for the final partitioning results generated by the unit BASH.

(3) Then the BASH generates the final partitioning result using the algorithm described in Section 4.5. Finally, the Resource Manager will transmit the results back to the application Masters through the resource response message.

4.2 Parallel Reservoir Sampling Strategy

In PIY, we get the distribution of the intermediate result by running a novel sampling during the normal map processing. Our sampling is performed by some map tasks with higher priority. Therefore, when split operation in map stage finishes, the map tasks conducting sampling are processed preferentially. Obviously, there is tradeoff between the sampling overhead and the accuracy of the result. In our experiments, we find that by integrating sampling into 20% of map tasks, a sufficient accuracy approximation can be achieved. Our sampling strategy, namely Parallel Reservoir Sampling, simply PRS for short, is based on reservoir sampling algorithm[15]. PRS runs by invoking the class org.apache.hadoop.mapreduce.lib.InputSampler and overloading the SplitSampler method.

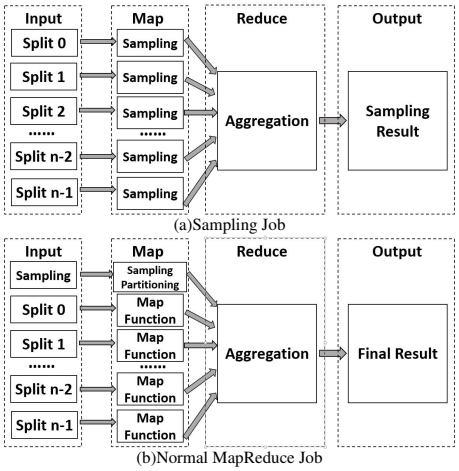


Figure 2: The Process of Parallel Reservoir Sampling

The main idea of PRS is described as follows. PRS builds one reservoir for each 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 stored elements with it 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 the reduce function. The process of PRS is shown as Figure 2(a). When the sampling tasks of some splits are finished, their corresponding normal user-defined map function are executed successively. Thus, our PRS is integrated into the normal map processing. As is shown in Figure 2(b), when all samplers are complete, the sampling result will be aggregated and transmitted to GDFT model in the NameNode which decides the sampling partition. 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 reduce stage in native hadoop because the decision of sampling partition introduces overhead. However, this overhead is negligible based on our experimental results shown in Section 5.

4.3 The Capacity Monitor

The Hadoop cluster is ofen heterogenous. To get the computing capacity of each DataNode, we designed a special model, namely, Capacity Monitor, in each DataNode. To decrease the extra overhead dramatically, the Capacity Monitor in each DataNode keeps monitoring the implementation of the sampled input data when the sampling function begins to run, and gets its consuming volume $Volume(con)_{id}(1 \leq id \leq m)$ during a period of time Δt, here m denotes the number of DataNode in cluster. Then

we can calculate the capacity value of the id^{th} DataNode, CV_{id}, by following formula (3). Capacity Monitors sends the capacity values of the DataNode to the PIY in NameNode through the heartbeat message.

$$CV_{id} = Volume(cons)_{id}/\Delta t \quad (3)$$

4.4 Network Traffic in Shuffle Phrase

As the bandwidth is the scarce resource in networks, the shuffle phase has become the bottleneck of MapReduce due to its large amount of network traffic. As is known, there are many Reducers in cluster, especially in Datacenter Network (DCN). In addition, many DataNodes act as both Mapper and Reducer[12]. If we can stay as many as intermediate <key,value> pairs on these DataNodes by the partition method in shuffle phase, it also furthest decrease the network traffic[5]. It's assumed that there are many <key,value> pairs corresponding to a special key on those DataNodes simultaneously. The BASH algorithm will find the DataNode that contains the maximum amount of these pairs, and then transmits all the pairs corresponding to the key to this DataNode. Our experimental result proves this method could decrease the network traffic in shuffle phase by up to 19.11%.

4.5 BASH Algorithm

In this section, we describe our proposed algorithm named BASH that comprehensively considers the load **BA**lance among all Reducers, network traffic in **S**huffle and the **H**eterogeneity of Hadoop cluster. As shown in algorithm 3, there are three steps in BASH. First, it minimizes intermediate data transmission in shuffle phase. Second, it gets the data volume that each reducer should process according to their computing capacity. Finally, to balance the load among Reducers, BASH partitions intermediate data to each Reducer using Approx-Subset-Sum algorithm.

We assume that there are *k* distinct <key,value> pairs corresponding to various keys, and *r* Reducers in cluster. key_dest_i records the serial number of the destination Reducer that will process the <key,value> pairs corresponding to key_i, and all key_dest_i consist of the array key_dest[1,...,k]. Lines 1-3 initialize all key_dest_i with -1, which means all <key,value> have not been partitioned. The array RS[1,...,r] records the volume of data that should be processed by special Reducers. CV[1,...,r] records the computing capacity of each Reducer, the value of CV_i can be obtained by formula (3). Sum_CV records the total computing capacity value of all Reducers. Lines 4-7 initialize all RS_i 1≤i≤r, and compute the Sum_CV.

Lines 8-17 reduce the amount of network traffic in shuffle phase. As described in section 4.4, we focus on the DataNodes who act as both Mapper and Reducer. For each key_i $(1 < i < k)$ on these Reducers, BASH 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 corresponding to the key_i is the maximum, and then set

Algorithm 3 BASH Algorithm

Input: **k**: the number of <key, value> pairs; **r**: the number of Reducer; **key_size[1,..,k]** : the data volume of all <key,value> pairs ; **CV[1,...,r]** : the computing capacity value of every Reducer; **Sum_CV**: the total capacity value of all Reducers; ε : approximation parameter; **RS[1,..,r]**: the volume of the data that have been determined to be processed in every Reducer; **T[1,...,r]**: the remaining capacity of every Reducer; **Total_Size**: the total volume of all <key,value> pairs produced by all Mappers.

Output: **key_dest[1,...,k]**: A array indicating the destination Reducer of every key;

1: **for** i = 1 to k **do**
2: $key\_dest_i = -1$;
3: **end for**
4: **for** i = 1 to r **do**
5: $RS_i = 0$;
6: $Sum\_CV = Sum\_CV + CV_j$;
7: **end for**
8: **for** each Reducer $R_j (1 \leq j \leq r)$ **do**
9: **if** R_j is also a Mapper **then**
10: **for** every key_i 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: **end if**
15: **end for**
16: **end if**
17: **end for**
18: **for** j = 1 to r **do**
19: $T_j = Total\_Size * (CV_j/Sum\_CV) - RS_j$;
20: **end for**
21: **for** j=1 to r **do**
22: Z^* = $Approx\_Subset\_Sum(key\_size[1,...,k], T_j, \varepsilon)$;
23: Set key_dest of the keys which composing Z^* to the sequence number of $Reducer_j$;
24: **end for**
25: **for** i = 1 to k **do**
26: **if** $key\_dest_i == -1$ **then**
27: key_dest_i = the sequence number of the strongest capacity Reducer;
28: **end if**
29: **end for**
30: **return** $key\_dest[1,...,k]$;

this Reducer as the destination Reducer of key_i. Line 13 updates the volume of the data that should be processed on this Reducer. Here, the array key_size[1,...,k] records the data volume of all <key,value> pairs. Lines 18-20 get the remaining capacity of each Reducer, which is denoted as array *T*. Here remaining capacity means the extra data volume that one Reducer can process. The Total_Size denotes the total volume of experimental data set. Total_Size * (CV_j/Sum_CV) means the total data volume that the $Reducer_j$ should process according to its computing capacity.

Using Approx_Subset_Sum algorithm, lines 21-29 balance the load among all Reducers by partitioning the intermediate data based on Reducer's computing capacity. In lines 21-24, BASH partitions intermediate data to each Reducer and records the destination Reducer of each key into the array key_dest. We can get these value through the GDFT in PIY. As we describe in Section 3, the Approx_Subset_Sum algorithm only gets an approximate result that does not reach the target value. Therefore, the amount of data that is partitioned to each Reducer could not reach its capacity value. This generates some trivial datasets that are not partitioned finally. In lines 25-29, BASH assigns these trivial datasets to the Reducer with the strongest computing capacity.

5 Evaluation

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

5.1 Experimental Environment

In our experiments, we set up two Hadoop clusters, one is homogeneous, and the other is heterogeneous. Our Hadoop homogeneous cluster consists of 60 physical machines installed with Ubuntu 12.04(KVM as the hypervisor) with 16 core 2.53GHz Intel processors, 4G memory, and the 60 nodes connected through a single switch, the network bandwidth is 1Gbps. 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 processors, 4G memory. The second type contains 20 machines with 4 core 3.3GHz Intel processors, and 8G memory. The third type contains 10 machines with 2.4GHz Intel processors, and 2G memory. The other configurations are same in the homogeneous cluster.

Table 1: Jobs with Different Sampling Methods

Sampling Method	Time(s)	Sample File Size(MB)	$Accu_{appro}$
Random	2.8	1.2	307889
TopCluster	2.5	1.3	142728
PRS	2.6	5	97335

In this section, we evaluate PIY by running different type of bench-marks in homogeneous and heterogenous Hadoop cluster, respectively. In order to ensure accuracy, we performed each group of experiments at least 10 times and took the mean value as the final result so as to reduce the influence of the environment.

5.2 Accuracy of the Sampling Method

We compare our PRS with the other two sampling methods: the random sampler used in native Hadoop and TopCluster [3]. We run three different samplers on a 10GB real-word data sets from the full English Wikipedia archive, which contains 50000 keys. We measure the sampling approximation by formula(4), where x_i^{appro} and x_i^{real} denote the sampling and real frequency of tuple corresponding to the key *i*, respectively. The smaller value of $Appro_{sampl}$, the better. All three methods sample 20% of input splits and 1000 keys from each split. Table 1 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. The better accuracy can be realized if the sampling result is larger.

From Table 1 we can see that the approximation of our PRS is 97335, which is better than the other two sampling methods. This is also visualized in Figure 3 for the top 1000 large keys in the data. Note that the sampling approximation of TopCluster is fairly accurate on the large keys which are at the beginning of the curve, representing their frequency are relatively large, but terrible on the keys whose frequency is lesse than 10^3. The reason is that TopCluster assumes the distribution of small keys are in accord with the large keys, and this assumption can be misleading when there are a large number of small keys in the data.

$$Appro_{sampl} = \sqrt{\frac{\sum_{i=1}^{n}(x_i^{appro} - x_i^{real})^2}{n}} \qquad (4)$$

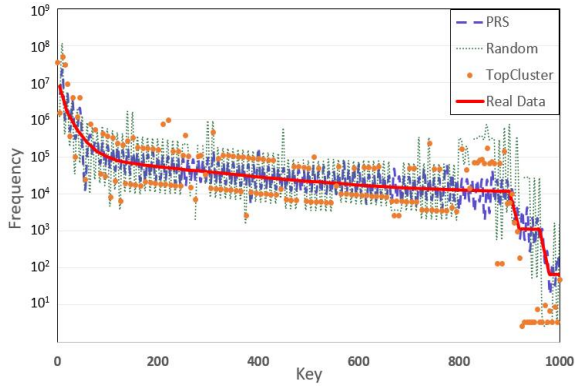


Figure 3: Comparison of three sampling methods in Grep

5.3 Load Balance among Reducers When Running Sort Benchmark

One of majar motivations for PIY is to balance loads among Reducers when data skew happens. Therefore, in this section, we evaluate PIY by running Sort benchmark, which represents reduce-input-heavy job, to process the input data with various data skew degrees. In this paper, the load balancing and data skew degree are measured by the coefficient of variation, which is represented as COV. The smaller COV, the better. Figure 4(a) and Figure 4(b) show COV when running sort benchmark based on 10 GB of synthetic data in homogeneous and heterogenous cluster respectively. We generate a 10GB synthetic data set following Zipf [1] distributions with varying δ parameters from 0.2 to 1.2 to control the degree of the skew.

As Figure 4(a) shows, the curves of Hadoop-Hash and SkewTune keep rising when the data skew rate increases, while the COV of PIY remains very low all the time. This can be explained as PIY partitions the intermediate data to all Reducers evenly in homogeneous cluster. The reason why SkewTune performs worse than PIY is that SkewTune can only repartition the input data of one straggler at a time, it could 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 unbalance loads seriously when data skew happens, which makes it the worst performance in Figure 4 (a).

From Figure 4(b), we find the same results as in Figure 4(a).In addition, while the value of PIY are almost unchange, the values of Hadoop-Hash and SkewTune at most of data skew rates are higher than that in homogeneous cluster, and this trend is more obvious when the data skew rate increases. In other words, the optimization degree of PIY in load balance in heterogeneous cluster is much more than that in homogenous cluster. Beside the reasons we have described in the prior paragraph, the consideration of heterogeneity of PIY made a greater contribution in load balance among Reducers.

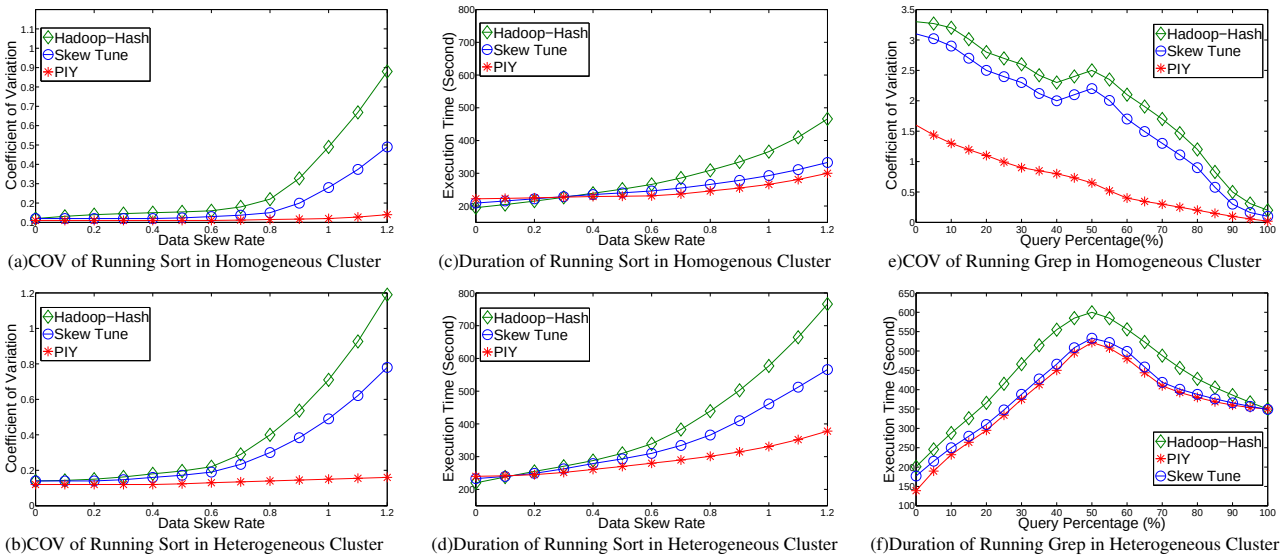
5.4 Execution Time of Sort Benchmark

Figure 4 also shows the execution time of the experiments we have described in Section 5.3. The curves in Figure 4(c) shows the results in the homogenous cluster. We can see PIY 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, PIY does not perform satisfactorily. The reason is that when the data skew degree is low, e.g. less than 0.28 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 produces small overhead on moving unprocessed data of the slower tasks because there are few stragglers in this scenario, and this leads to its execution performance behinds the Hadoop-Hash. Furthermore, PIY consumes the longest execution time because of its extra overhead produced by sampling data and making partition decision in map phase. However, as data skew degree increases, the optimization, which is achieved through balancing load among Reducers using approx_sum_subset algorithm, gradually offsets the time spent by the extra overhead. Therefore, PIY consumes the lowest execution time. Consequently, compared with Hadoop-Hash and SkewTune, PIY achieves the average improvements in execution time by 16.39% and 4.71%, respectively, and the maximum improvements reach 35.62% and 9.90%, respectively, when the data skew rate is 1.2.

Figure 4(d) shows the fully adaption of PIY to heterogeneous cluster. PIY is also the fastest one in most cases. The threshold, less than which the performance of PIY is worse than the other two, is 0.17 and it less than 0.28 in Figure 4(c). On average, PIY can perform 29.4% and 14.84% faster than Hadoop-Hash and SkewTune, respectively. Specially, when the data skew is set to 1.2, the improvement is up to 50.65% and 24.54%, respectively. These values demonstrate that the degree of improvements PIY makes is more obvious in heterogeneous cluster than that in homogeneous cluster because it considers the computing capacity of every node during partitioning.

5.5 Grep Benchmark Testing

To evaluate the performance of PIY when it deals with the reduce-input-light applications, we run Grep, which is a light job for reduce, in heterogeneous cluster. 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 use



(a)COV of Running Sort in Homogeneous Cluster (c)Duration of Running Sort in Homogenous Cluster e)COV of Running Grep in Homogeneous Cluster

(b)COV of Running Sort in Heterogeneous Cluster (d)Duration of Running Sort in Heterogeneous Cluster (f)Duration of Running Grep in Heterogeneous Cluster

Figure 4: Evaluation of PIY running benchmarks In Hadoop Clusters

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%. Figure 4(e) and (f) show that PIY gets the best performance of COV and job execution time at all time due to the accuracy of PRS and the consideration of heterogeneity. Specially, in Figure 4(e), PIY gets the best COV when the query percentage is lower. This is because the PRS in PIY 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 gap rapidly closes.

5.6 Optimization In Shuffle Phase

To verify that the BASH algorithm used by PIY can decrease the amount of data transmission in shuffle phase, we record the execution of each phase in MapReduce of Sort job in the heterogenous cluster. Without losing generality, we illustrate the duration time when δ is 0.8 in Figure 5(a). The native Hadoop starts the shuffle tasks when 5% map tasks finish, therefore, we divided MapReduce into 4 phases, which are represented as Map(Seperate), Concurrent Map and Shuffle, Shuffle(Separate), and Reduce. Concurrent Map and Shuffle denotes 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, whose fill patterns are red in Figure 5, equals to the sum of Shuffle(Separate) and "Concurrent Map and Shuffle". Specially, because PIY executes PRS in Map phase, its duration of Map phase should contain additional time costed by PRS, which is represented as Sampling in Figure 5. From Figure 5(a), we can see the duration in shuffle phase of PIY is 105+7=112 seconds, which is less than the SkewTune (107+19=126) and Hadoop-Hash (105+39=144), the improvement are $\frac{126-112}{126} = 11.11\%$ and 22.22%.

Through plenty of experiments, we can see that compared with Hadoop-Hash and SkewTune, the improvement degree PIY achieves in 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 can run at 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 which nodes are Reducers according to the computing resource (container in Yarn) in each reducer. The results are shown in Figure 5(b). We can easily find the duration in shuffle phase of PIY is 89+9=98 seconds, which is much less than 119 seconds for SkewTune, 143 seconds for Hadoop-Hash, the improvement is up to $\frac{119-98}{119} = 17.65\%$ and 31.47%, which are larger than Figure 5(a). This can be explained as with the number of Reducers increases, the BASH algorithm finds much input data whose map and reduce tasks are able to be scheduled to the same DataNode. This results in decreasing the amount of data transmission in 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 PIY are reduced to 12.35% and 19.11%,

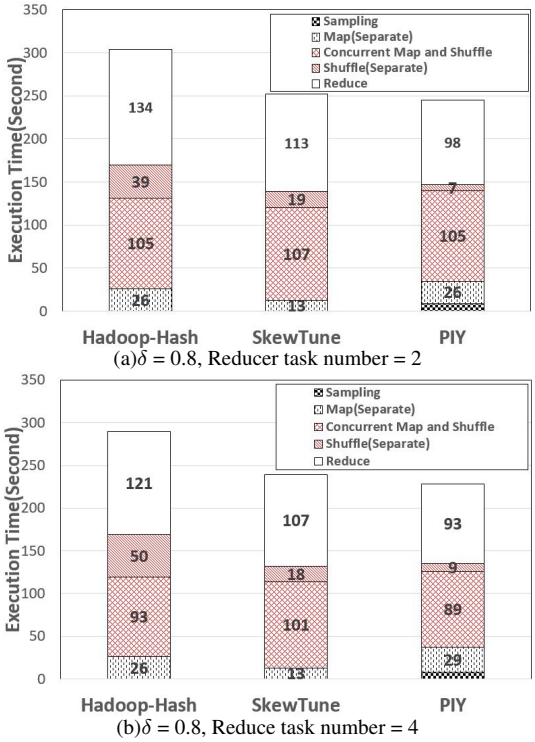


Figure 5: The Execution Time Of Each Phase

respectively. How to find the optimal Reducer number is a problem about the tradeoff between the computing parallelism degree and the network transmission amount, and this is our future works on PIY.

5.7 PIY in parallel image processing

Table 2: the size of sample images

image name	size(in bytes)
CARTOSAT-1	1342552576
CARTOSAT-2A	4259355002
CARTOSAT-2B	9204661322

With the need of processing large-sized images increases rapidly, parallel image processing technologies, such as MapReduce, are widely used to shorten the execution time. In this section, we present the experiments conducted for images with large sizes approximately from 1.3 Gigabytes to 9.1 Gigabytes the Chinese Remote Sensing (IRS) satellite series. The sample data sets are shown in Table 2.

We conduct histogram [7] operation on native Hadoop, HIPI [14], which is a open-source Hadoop Image Processing Interface, and PIY. Compared with native Hadoop, HIPI processes image without requiring the additional coding because it implements Java Image Processing Library. His-

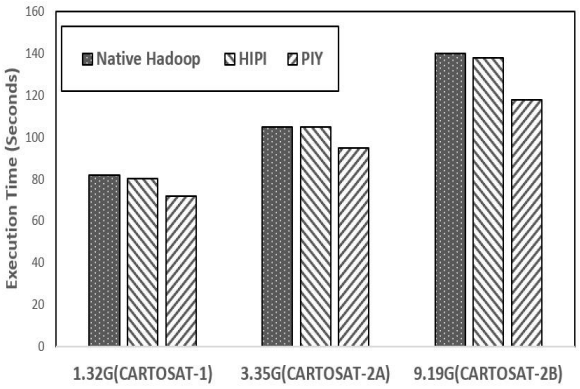


Figure 6: Execution Time Of Histogram

togram operation counts the frequency of the pixel intensity in an entire image, which is similar to counting the words in the 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 the aggregation of the collected numbers from the map functions. To increase the amount of input data in reduce phase, we add TeraSort operation in histogram and finally output the pixel intensity result in 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, one first type node acts as master, 2 nodes for each of other two types act as slaves.

As is shown in Figure 6, PIY gets the shortest execution time when processing all 3 different large-sized images. The execution time is reduced by 11.2%. The reasons are described as follows. First, the distribution of the frequency of the pixel intensity in an large-sized image is not even in general, i.e., the pixel intensity values are skew. Different with native Hadoop and HIPI, PIY considers the data skew by balancing the loads on Reducers. Second, the PRS sampling method helps PIY to realize more accurate distribution of the pixel intensity than the other two frameworks. Third, heterogeneity consideration helps PIY achieve the fastest process speed.

6 Conclusion

This paper proposes PIY to mitigate data skew in MapReduce system. Using the parallel reservoir sampling method we proposed, PIY achieves the distribution of intermediate data accurately with negligible overhead. PIY tries to reduce the network traffic in shuffle phase by decreasing the transmission amount of data on those nodes acting as both Mapper and Reducer. PIY also considers the heterogeneity of the computing resource when balanc-

ing load among Reducers. Performance evaluation in both synthetic and real workloads demonstrates that the resulting performance improvement is significant. Compared with some other popular strategies, the improvement PIY achieved reaches 35.62% and 50.65% in homogeneous and heterogeneous clusters, respectively. PIY can also be used in parallel imagine processing to reduce the execution time.

ACKNOWLEDGMENT

This work was supported in part by National Natural Science Foundation of China(No.61272353, No.61428201) and 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).

References

[1] L. A. Adamic and B. A. Huberman. Zipfs law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[2] Q. Chen, J. Yao, and Z. Xiao. Libra: Lightweight data skew mitigation in mapreduce. *IEEE Transactions on parallel and distributed systems*, 26(9):2520–2533, 2015.

[3] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 522–533. IEEE, 2012.

[4] H. H. Hong Zhang and L. Wang. MRapid: An efficient short job optimizer on hadoop. In *the 31st IEEE International Parallel Distributed Processing Symposium (IPDPS)*. IEEE, 2017.

[5] H. Huang, L. Wang, B. C. Tak, L. Wang, and C. Tang. Cap3: A cloud auto-provisioning framework for parallel processing using on-demand and spot instances. In *Cloud Computing (CLOUD), 2013 IEEE Sixth International Conference on*, pages 228–235. IEEE, 2013.

[6] 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 *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 17–24. IEEE, 2010.

[7] J. N. Kapur, P. K. Sahoo, and A. K. Wong. A new method for gray-level picture thresholding using the entropy of the histogram. *Computer vision, graphics, and image processing*, 29(3):273–285, 1985.

[8] C. Kumar. Approximation algorithm project. *arXiv preprint arXiv:1401.2393*, 2014.

[9] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 75–86. ACM, 2010.

[10] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 25–36. ACM, 2012.

[11] J. Liu, F. Liu, and N. Ansari. Monitoring and analyzing big traffic data of a large-scale cellular network with hadoop. *IEEE Network*, 28(4):32–39, 2014.

[12] V. Subramanian, H. Ma, L. Wang, E.-J. Lee, and P. Chen. Rapid 3d seismic source inversion using windows azure and amazon ec2. In *Proceedings of the 2011 IEEE World Congress on Services*, SERVICES '11, pages 602–606. IEEE, 2011.

[13] V. Subramanian, L. Wang, E.-J. Lee, and P. Chen. Rapid processing of synthetic seismograms using windows azure cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 193–200. IEEE, 2010.

[14] C. Sweeney, L. Liu, S. Arietta, and J. Lawrence. Hipi: a hadoop image processing interface for image-based mapreduce tasks. *Chris. University of Virginia*, 2011.

[15] J. S. Vitter. Random sampling with a reservoir. *ACM Transactions on Mathematical Software (TOMS)*, 11(1):37–57, 1985.

[16] L. Wang, S. Lu, X. Fei, A. Chebotko, H. V. Bryant, and J. L. Ram. Atomicity and provenance support for pipelined scientific workflows. *Future Generation Computer Systems*, 25(5):568–576, 2009.

[17] Z. Xiao, W. Song, and Q. Chen. Dynamic resource allocation using virtual machines for cloud computing environment. *IEEE transactions on parallel and distributed systems*, 24(6):1107–1117, 2013.

[18] Y. Xu, W. Qu, Z. Li, Z. Liu, C. Ji, Y. Li, and H. Li. Balancing reducer workload for skewed data using sampling-based partitioning. *Computers & Electrical Engineering*, 40(2):675–687, 2014.

[19] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Osdi*, volume 8, page 7, 2008.

[20] H. Zhang, L. Wang, and H. Huang. Smarth: Enabling multi-pipeline data transfer in hdfs. In *Parallel Processing (ICPP), 2014 43rd International Conference on*, pages 30–39. IEEE, 2014.