

# OPTIMA: On-Line Partitioning Skew Mitigation for MapReduce with Resource Adjustment

Zhihong Liu<sup>1</sup> · Qi Zhang<sup>2</sup> · Raouf Boutaba<sup>2</sup> ·  
Yaping Liu<sup>3</sup> · Baosheng Wang<sup>1</sup>

Received: 2 May 2015 / Accepted: 18 December 2015  
© Springer Science+Business Media New York 2016

**Abstract** Partitioning skew has been shown to be a major issue that can significantly prolong the execution time of MapReduce jobs. Most of the existing off-line heuristics for partitioning skew mitigation are inefficient; they have to wait for the completion of all the map tasks. Some solutions can tackle this problem on-line, but will impose an additional overhead by repartitioning the workload of overloaded tasks. In this paper, we present OPTIMA, an on-line partitioning skew mitigation technique for MapReduce. OPTIMA predicts the workload distribution of reduce tasks at run-time, leverages the deviation detection technique to identify the overloaded tasks and pro-actively adjusts resource allocation for these tasks to reduce their execution time. We provide the upper bound of OPTIMA in time complexity, while allowing OPTIMA to perform totally on-line. Through experiments using both real and synthetic workloads

---

Currently, Zhihong Liu is with University of Waterloo as a visiting student.

---

✉ Raouf Boutaba  
rboutaba@uwaterloo.ca

Zhihong Liu  
zhliu@nudt.edu.cn

Qi Zhang  
q8zhang@uwaterloo.ca

Yaping Liu  
ypliu@nudt.edu.cn

Baosheng Wang  
bswang@nudt.edu.cn

- <sup>1</sup> College of Computer, National University of Defense Technology, Changsha, Hunan, China
- <sup>2</sup> David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada
- <sup>3</sup> Science and Technology on Parallel and Distributed Processing Laboratory, National University of Defense Technology, Changsha, Hunan, China

running on an 11-node Hadoop cluster, we have observed OPTIMA can effectively mitigate the partitioning skew and improved the job completion time by up to 36.73 % in our experiments.

**Keywords** MapReduce · Partitioning skew · Resource allocation · Scheduling

## 1 Introduction

With the emergence of the cloud computing paradigm and development in web applications, scientific computing and sensor networks, huge amounts of data are being generated continuously. Thus, efficiently and promptly analyzing these large datasets so as to extract relevant information for decision making is becoming an issue of vital importance. MapReduce [8], due to its remarkable advantages in simplicity, robustness, and scalability, has gained significant popularity as a big data analysis framework. Many large companies, such as Amazon, Facebook, and Yahoo!, have been using MapReduce routinely to process large volumes of data on a daily basis.

While highly successful, the current implementations of MapReduce still suffer from an issue known as *partitioning skew*. In particular, Apache Hadoop MapReduce [32], one of the most popular implementations of MapReduce, uses a hash function to partition the intermediate data among the reduce tasks. While the goal of using the hash function is to evenly distribute workload to each reduce task, in reality this goal is rarely achieved. For example, Zacheilas and Kalogeraki [37] have shown the existence of skewness in a MapReduce application based on real-world social graph data; their experiments show that the largest partition can be more than five times larger than the smallest partition in the same job.

The skewed workload distribution among tasks can negatively impact both job performance and resource utilization. First, the completion time of a MapReduce job is determined by the execution time of its slowest task. The run-time variation of parallel tasks, which is caused by the skewed workload distribution, may prolong the execution of the entire job. Second, skewed workload distribution gives rise to variation in resource requirements. As a result, machines that run tasks with heavy workload may experience resource contention, while machines with less data to process may incur resource idleness.

The partitioning skew problem in MapReduce has been extensively investigated in recent years. One strategy is to rebalance the key distribution among reduce tasks [11, 12, 16, 29]. However, this catalog of approaches will lead to a synchronization barrier that can slow down job execution. This is because in order to obtain key distribution statistics, they have to either wait for the completion of map tasks or perform sampling before the execution of the job. Another strategy is to reassign more work to more powerful nodes [21]. This strategy identifies straggling tasks, often called *stragglers*, based on the task's remaining time, and then repartition the unprocessed workloads of stragglers to the nodes that exhibit better performance. However, the overhead due to repartitioning (as reported in [21], approximately

30 s overhead is incurred) can be quite large for small jobs. A final strategy is to run additional speculative tasks on other machines [38]. This technique is prominently used in production clusters such as Facebook and Microsoft Bing [1]. It monitors the progress of every task, and spawns redundant copies for tasks at a slow progress rate, hoping that the replica tasks will run faster than the originals. However, being agnostic to the correlation between task workload and progress rate, replication-based approaches may waste resources.

Motivated by the limitations of existing approaches, in our prior work [25], we proposed DREAMS, a framework that provides run-time partitioning skew mitigation. DREAMS leverages historical records to construct profiles for each type of jobs. Based on the profile, DREAMS then allocates the right amount of resources to reduce tasks in order to decrease the variation of their execution time. The main drawback of DREAMS is that it requires job profiling. Even though job profiling is reasonable for many jobs that are executed repeatedly in today's production clusters [33], building a job profile requires executing a large set of benchmarks with various task resource allocations, which is both expensive and time consuming. Moreover, DREAMS cannot handle the partitioning skew problem of jobs that have not been encountered before. This places a significant constraint on its applicability.

In this paper, we present OPTIMA, an **O**n-line **P**artitioning skew **m**i**T**iga-tion technique for **M**apReduce with resource **A**djustment. In contrast with DREAMS, OPTIMA does not require job profiling and eliminates the dependency on the task performance model. This not only eliminates the limitation of applicability to only routine jobs in DREAMS, but also allows the solution to be carried out in an on-line manner. In particular, we develop a low complexity on-line partition size prediction model. Further, we propose a data skew detection algorithm that can identify the *overloaded tasks*, which are the reduce tasks with extremely large workload, in linearithmic time. Finally, we propose a scheduling algorithm that adjusts resource allocation to the overloaded tasks with the goal of reducing the variation of task running time and accelerating job completion. Through experiments using both real and synthetic workloads running on an 11-node Hadoop cluster, we show that OPTIMA can effectively mitigate negative impact of partition skew, thereby improving job performance by up to 36.73 %.

The rest of this paper is organized as follows. Section 2 provides the background and motivations of our work. We describe the system architecture of OPTIMA in Sect. 3. Section 4 details the design of OPTIMA. Section 5 presents the results of experimental evaluation. Finally, we summarize existing work related to OPTIMA in Sect. 6, and draw our conclusion in Sect. 7.

## 2 Motivation

This section provides an overview of Hadoop MapReduce and elaborates the partitioning skew issue therein motivating our study.

## 2.1 Hadoop MapReduce

MapReduce [8] is a parallel computing model for large-scale data-intensive computations. In MapReduce, the input data is split into uniform sized data chunks (e.g. 64 or 128 MB), which are stored in a distributed file system across the cluster nodes. Each of these data chunks is called an *inputsplit*. There are two types of tasks in MapReduce, namely map and reduce tasks. A map task takes one *inputsplit* as input, applies the user-defined map function on its input and generates a sequence of key-value pairs called intermediate data. A hash function is then used to divide the intermediate data into a number of partitions and distribute them across reduce tasks. A reduce task takes one partition ( i.e. the intermediate key-value pairs corresponding to the same hash values) as input and performs the reduce function on its partition to generate the final output.

In a nutshell, MapReduce adopts a “divide-and-conquer” approach for data-intensive computations. In the map stage (*a.k.a.* the “divide” phase), the processing of a job is divided to a number of independent sub-problems, and each sub-problem is processed by a map task. Subsequently in the reduce stage (*a.k.a.* the “conquer” phase), the output of all the sub-problems is aggregated by a number of reduce tasks, thereby generating the final output for the job. By breaking down a data-intensive job into a large number of small tasks and executing them in parallel across multiple machines, MapReduce can significantly reduce the job running time.

Currently, the de facto implementation of MapReduce is Apache Hadoop MapReduce (MRv1) [13]. It consists of a *JobTracker* that is responsible for task scheduling and a number of *TaskTrackers* that are responsible for launching and allocating resources for tasks. To do so, the *TaskTracker* launches a Java Virtual Machine (JVM) that executes the corresponding map or reduce task. MRv1 adopts a slot-based resource allocation scheme where each machine is divided into identical *slots* that can be used to execute tasks. The number of map slots and reduce slots determines respectively the maximum number of map tasks and reduce tasks that can be scheduled on the machine at a given time.

Due to many inadequacies experienced in MRv1, the next generation of the Hadoop compute platform, YARN [32], has been developed. Compared to MRv1, YARN introduces *ResourceManager* and *ApplicationMaster*, and divides the function of scheduling to two parts: the *ResourceManager* is responsible for allocating resources to applications subject to constraints of capacities, fairness, *etc.*; the *ApplicationMasters* has the responsibility of negotiating resources from the *ResourceManager* and assigning the obtained resources to its tasks. In particular, YARN deprecates the slot-based resource management approach. Instead, a more flexible resource unit called *container* is adopted, which provides specific resource accounting and enforces the resource limit on the task running within it.

Nevertheless, in both MRv1 and YARN, the schedulers assume each reduce task has uniform workload and resource consumption, and therefore allocate identical resources to each reduce task. In the presence of partitioning skew, this scheduling scheme can cause a variation in task running time and degradation in resource utilization. More specifically, the reduce tasks with heavy workload run slowly because the resources allocated to them are limited by the “slot” or container size,

whereas reduce tasks with light workload tend to under-utilize the resources allocated to them. In both cases, the resulting resource allocation is inefficient, thus prolonging the job completion time.

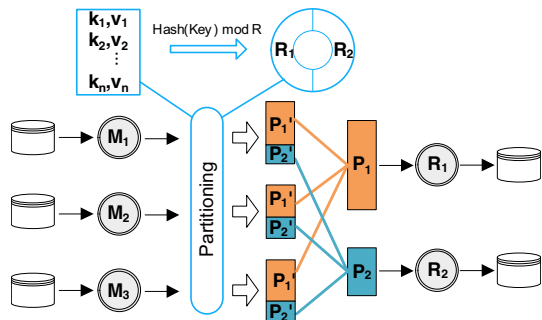
### 2.2 Partitioning Skew

In the current implementations of MapReduce, the output of map tasks are distributed among reduce tasks via a hash function  $\text{Hash}(\text{HashCode}(\text{key}) \bmod \text{number of reducers})$ . Typically, this hash function can provide load balancing if the key frequencies and the size of key-value pairs are uniformly distributed. However, this may fail with skewed data. For example in the InvertedIndex application, the hash function partitions the intermediate data based on the words appeared in the *inputsplit*. Therefore, reduce tasks which take popular words as keys will be assigned a larger workload. As shown in Fig. 1, after partitioning the intermediate data by the hash function.  $P_1$  and  $P_2$  are distributed to  $R_1$  and  $R_2$  respectively. However, since  $P_1$  is larger than  $P_2$ , workload imbalance between  $R_1$  and  $R_2$  is incurred. The partitioning skew is prevalent in MapReduce applications, and it can be caused by following reasons [11]:

1. *skewed key frequencies*: Some keys occur more frequently in the intermediate data, causing those reduce tasks that process these popular keys to become overloaded.
2. *skewed tuple sizes*: In applications where the sizes of values in the key-value pair vary significantly, uneven workload distribution may arise.
3. *skewed execution times*: Typical in scenarios where processing a single, large key-value pair may require more time than processing multiple small pairs. Even if the overall data size per reduce task is the same, the execution times of reduce tasks may be different.

We focus on the *skewed key frequencies* and *skewed tuple sizes*, as they are commonly seen in MapReduce jobs and have a significant impact on the job completion time. It is shown by recent works [11, 12, 16, 29, 37] that solving the partitioning skew problem is not trivial. Many solutions have been proposed. However, most of the existing approaches tackle this problem using off-line

**Fig. 1** MapReduce programming model



heuristics, either by waiting for all the map tasks to finish or sampling in advance, so as to estimate the reducer’s workload distribution and then balance the workload among reduce tasks. The authors of [15] demonstrate that by starting the shuffle phase after all map tasks are completed, the overall job completion time will be prolonged. Admittedly, Skewtune [21] tries to solve this problem in an on-line fashion by repartitioning the straggling tasks to fully utilize the nodes in the cluster. However, achieving this goal incurs additional overhead due to both adaptive partitioning and reconstructing the final output by concatenation. Our previous work, DREAMS [25], provides run-time partitioning skew mitigation without using load rebalancing and repartitioning. However, it relies on job profiles which limits its applicability to routing jobs only. Therefore, in this work we propose an on-line partitioning skew mitigation technique, which detects the overloaded tasks at run-time and pro-actively adjusts resource allocation for these tasks to reduce their execution times. This approach gets rid of job profiles and preforms totally on-line, and at the same time effectively mitigates the negative impact of partitioning skew.

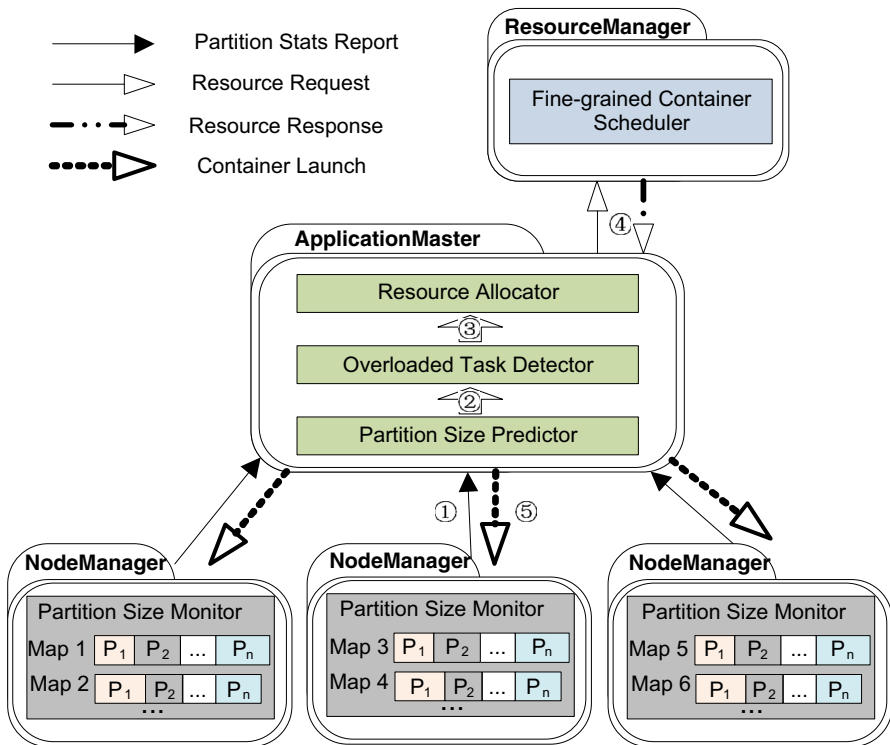


Fig. 2 Architecture of OPTIMA

### 3 System Architecture

This section describes the design of our proposed resource allocation framework called OPTIMA. The architecture of OPTIMA is shown in Fig. 2. There are five main components: *Partition Size Monitor*, running at the NodeManager; *Partition Size Predictor*, *Overloaded Task Detector* and *Resource Allocator*, running at the ApplicationMaster; *Fine-grained Container Scheduler*, running at the ResourceManager. Specifically, each *Partition Size Monitor* records the statistics of intermediate data generated by each map task at run-time and sends them to the ApplicationMaster through heartbeat messages. The *Partition Size Predictor* collects the partition size reports from NodeManagers and predicts the partition size of every reduce task for this job. Based on the estimated workload of reduce tasks, the *Overloaded Task Detector* then detects the overloaded reduce tasks. The *Resource Allocator* determines the amount of resources to be allocated to each reduce task based on the outliers detection results. Lastly, the *Fine-grained Container Scheduler* is responsible for scheduling resources among all the ApplicationMasters in the cluster, which is based on scheduling policies such as Fair scheduling [14] and Dominant Resource Fairness (DRF) [10]. Note that the scheduler in original Hadoop allocates containers of identical size to all reduce tasks (and similarly, all map tasks). We have modified the original scheduler to support fine-grained container scheduling where tasks can request containers of different sizes.

The workflow of resource allocation mechanism used by OPTIMA consists of 5 steps:

1. After the ApplicationMaster is launched, it schedules all the map tasks first and then ramps up the reduce task requests slowly according to the *slowstart* setting. During their execution, each *Partition Size Monitor* records the size of intermediate data produced by map tasks. It then sends the statistics to the ApplicationMaster through *heartbeat* messages which are used to monitor the status of tasks in Hadoop.
2. Upon receiving the partition size reports from the *Partition Size Monitors*, the *Partition Size Predictor* predicts the size of each partition using our proposed prediction model.
3. After the estimated size of each reduce task is known, the *Overloaded Task Detector* identifies overloaded reduce tasks. It then computes the resource allocation for each overloaded task. Afterwards, the ApplicationMaster sends resource requests to the ResourceManager.
4. The ResourceManager receives ApplicationMasters' resource requests through the heartbeat messages, and schedule free containers in the cluster to the corresponding ApplicationMasters.
5. Once the ApplicationMaster obtains new containers from ResourceManager, it assigns the matching container to its pending task, and finally launches the task.

## 4 OPTIMA Design

There are three main challenges that need to be addressed in OPTIMA: (1) predicting the reducer's workload distribution on-line with no priori knowledge of the map function and the input dataset; (2) detecting overloaded tasks in real-time without human invention and finally, (3) adjusting the resource allocations dynamically to accelerate the job completion. In the following sections, we shall describe our technical solutions for these challenges.

### 4.1 Predicting Partition Size

In order to cull the partitioning skew, the workload distribution among reduce tasks should be known in advance. Unfortunately, the sizes of partitions assigned to reduce tasks depend on the input dataset, the map function and the number of reduce tasks of the application. Even for MapReduce jobs that are routinely executed, different reducer's workload distributions can be produced if the input datasets are changed. As a result, the straightforward solution is to wait for the completion of all the map tasks or perform sampling before jobs start to gather the workload distribution statistics. However, in order to improve job running time, current MapReduce schedule overlaps the execution of map and reduce tasks by allowing reduce tasks to execute before the completion of all map tasks (e.g. the default slowstart setting is 5 %). In this case, if we gather the actual partition size after the completion of all map tasks and then schedule the reduce tasks, the job completion can be severely delayed. Therefore, it is necessary to predict the partition size on-line.

Since the input datasets of MapReduce applications in production clusters tend to be very large, monitoring the workload statistics in the level of key-value pairs is expensive. And HDFS [5] splits the large amount of data into small data chunks, which quite naturally creates a sampling space. We are motivated to use a small set of random data chunks to reveal some characteristics of the whole dataset in terms of workload distribution among reduce tasks. That is, we can analyse the pattern of intermediate data after a fraction of map tasks have completed, and then predict the reducer's workload distribution for the whole dataset.

Therefore, we perform a set of  $k$  measurements ( $j = 1, 2, \dots, k$ ) over time during the map phase, and collect the following two metrics  $(F^j, S_i^j)$ :

1.  $F^j$  is the percentage of map tasks that have been processed,  $j \in [1, k]$  and  $k$  refers to the number of collected tuples  $(F^j, S_i^j)$ . Note that each map task processes one inputsplit which is divided by HDFS, and each inputsplit has identical size of data (64, 128 MB, etc.). As a result,  $F^j$  is approximately equal to the fraction of the whole dataset that has been processed.
2.  $S_i^j$  is the size of the intermediate data generated by the completed map tasks for reduce task  $i$ . In our implementation, we have modified the reporting mechanism so that each map task reports this information to the ApplicationMaster upon map task completion.



Every time the percentage of completed map tasks ( $F^j$ ) changes, a new measurement of  $(F^j, S_i^j)$  is made. Hence, for each reducer  $i$ , there will be  $k$  tuples of  $(F^j, S_i^j)$  collected over time, where  $j = 1, 2, \dots, k$ . With this data, we use Ordinary Least Squares (OLS) [31] linear regression to determine the following equation for each reduce task  $i \in [1, N]$ :

$$S_i^j = \alpha_1 + \beta_1 \cdot F^j \quad j = 1, 2, \dots, k \tag{1}$$

where  $\alpha_1$  and  $\beta_1$  are the scaling factors which need to be obtained. We introduce an outer factor,  $\delta$ , which is the threshold to control our prediction model to stop the process of training and finalize the prediction. In practice,  $\delta$  can be the map completion percentage at which reduce tasks can be scheduled (e.g. 5 %). Every time a new map task has finished, a new training data is created. When the fraction of map tasks reaches  $\delta$ , we calculate the scaling factors  $(\alpha_1, \beta_1)$  and predict the size of partition for each reduce task  $i$ , even though not all of the map tasks are completed.

**Theorem 1** *The time complexity of the on-line partition size prediction model is  $O(k \cdot N)$ .*

*Proof* For each reduce task  $i \in [1, N]$ , the scaling factors can be provided by following equation:

$$\begin{pmatrix} \alpha_1 \\ \beta_1 \end{pmatrix} = (X^T X)^{-1} X^T Y, \tag{2}$$

where

$$X = \begin{pmatrix} 1 & F^1 \\ 1 & F^2 \\ \vdots & \vdots \\ 1 & F^k \end{pmatrix}, \quad Y = \begin{pmatrix} S_i^1 \\ S_i^2 \\ \vdots \\ S_i^k \end{pmatrix}$$

It takes  $O(k)$  to multiply  $X^T$  by  $X$ ,  $O(1)$  to compute the inversion of  $X^T X$ ,  $O(k)$  to multiply  $(X^T X)^{-1}$  by  $X^T$  and finally  $O(k)$  to multiply  $(X^T X)^{-1} X^T$  by  $Y$ . Hence, the computational complexity for predicting one task is  $O(k)$ . Assuming there are  $N$  reduce tasks, we need to preform prediction  $N$  times. Therefore, the total computational complexity for  $N$  tasks MapReduce jobs is  $O(k \cdot N)$ .  $\square$

We noticed that the load model in [29] can also be used by OPTIMA for partition size prediction. However, its sampling scheme is coupled with the partitioning plan, which is generated during the execution of map tasks, and as a result, this sampling scheme needs to be performed each time before the beginning of each job. In our case, since we do not need to modify the implementation of partitioning, our partition size prediction can be done entirely on-line.

### 4.2 Detecting Overloaded Tasks

By using the partition size prediction scheme presented in the previous section, we now can estimate the size of every reduce task at run-time. With the statistics of the reducer’s workload distribution in hand, however, determining which reduce tasks should be allocated more resources to is still a challenge. Here, we consider the reduce tasks which have extremely large workload as *overloaded tasks*. And these overloaded tasks are actually outliers in terms of the size of workload. Hence, we use a deviation detection algorithm [3] to identify them. Deviation detection is an outliers detection technique based on information theory, which neither makes assumption on the underlying statistic distribution of the data (e.g. statistic based outlier detection algorithm), nor requires human invention of specifying the metrical distance function (e.g. nearest neighbor based outliers detection algorithm). It tries to isolate the small minorities while maximizing the reduction in the deviation of the dataset. In order to apply the deviation detection algorithm in [3], we define the following concepts:

- the set of items  $I = \{P_i \mid i \in [1, N]\}$ (and its power set  $\mathcal{P}(I)$ );
- the dissimilar function  $\mathcal{D} : \mathcal{P}(I) \rightarrow \mathbb{R}_0^+$  be the variance of elements of the set;
- the cardinality function  $\mathcal{C} : \mathcal{P}(I) \rightarrow \mathbb{R}_0^+$  be the number of elements of the set, where  $I_1 \subset I_2 \Rightarrow \mathcal{C}(I_1) < \mathcal{C}(I_2)$  for all  $I_1, I_2 \subseteq I$ ;
- the smoothing factor for each  $I_j \subset I$ :

$$SF(I_j) = \mathcal{C}(I \setminus I_j) \cdot (\mathcal{D}(I) - \mathcal{D}(I \setminus I_j))$$

- We say that  $I_x \subset I$  is an exception set of  $I$  with respect to  $\mathcal{D}$  and  $\mathcal{C}$  if

$$SF(I_x) \geq SF(I_j) \quad \text{for all } I_j \subseteq I$$

Since many real-world datasets, such as frequency of word usage in English, ranking of world cities by population, ranking of number of ships built by all countries, etc., follow the Zipf’s law [20], where the less represented occurrences play a dominant role on the data distribution, we believe that the outliers in our circumstance are a small number of tasks but incur workload imbalance distinctly. As a result, we define the dissimilar function as the variance of the workload and the cardinality function as the count of tasks, which can cull the least number of tasks that most contribute to the dissimilarity. To demonstrate this scheme, consider an example that there are 4 reduce tasks and the size of each reduce task is predicted as 1, 2, 1 and 5 respectively. Thus the set  $I = \{1, 2, 1, 5\}$ . By computing the smoothing factor  $SF(I_j)$  for each candidate exception set  $I_j$ , we get:

$I_j$	$I \setminus I_j$	$\mathcal{C}(I \setminus I_j)$	$\mathcal{D}(I \setminus I_j)$	$SF(I_j)$
{ }	{1, 2, 1, 5}	4	3.59	0.00
{1}	{1, 2, 5}	3	4.33	-2.22

$I_j$	$I \setminus I_j$	$C(I \setminus I_j)$	$\mathcal{D}(I \setminus I_j)$	$SF(I_j)$
{2}	{1, 1, 5}	3	5.33	-5.22
{5}	{1, 2, 1}	3	0.33	9.78
{1, 2}	{1, 5}	2	8.00	-8.82
{2, 5}	{1, 1}	2	0.00	7.18
{1, 5}	{1, 2}	2	0.50	6.18
{1, 1, 5}	{2}	1	0.00	3.59
{1, 2, 5}	{1}	1	0.00	3.59
{1, 1, 2}	{5}	1	0.00	3.59

Therefore, the reduce task in the set  $I_j = \{5\}$  which has five units of workload can be qualified as the laggard. A simple strategy to solve this deviation detection problem is to iterate over all subsets of the universal set  $I$ , then compute the smooth factor for every subset to identify the subset with the biggest smooth factor as the set of the overloaded tasks. Since there are  $N!$  subsets of  $I$  when it contains  $N$  reduce tasks, the complexity of this simple strategy is at least  $O(N!)$ , even ignoring the complexity of computing  $SF(I_j)$  in each iteration step. Hence, it is inefficient for detecting the overloaded tasks for large dataset in real-time. This motivates us to innovate a fast method for overloaded outliers detection, which reveals linearithmic time complexity.

**Algorithm 1** Overloaded task detection algorithm

**Input:**  $I$  - set of the predicted sizes of all the reduce tasks, in particular,  $I = \{P_i \mid i \in [1, N]\}$

**Output:**  $I_x$  - Set of overloaded tasks.

- 1: sort the elements in  $I$  in decreasing order of the distance from the mean, say  $I' = \{E_1, E_2, \dots, E_N\}$
- 2:  $I'_0 = \phi$
- 3:  $I_x = \phi$
- 4:  $C_0 = N$
- 5:  $\mathcal{M}_0 = Mean(I')$
- 6:  $\mathcal{D}_0 = Variance(I')$
- 7:  $SF(I_x) = 0$
- 8: **for**  $j$  from 1 to  $N$  **do**
- 9:  $I'_j = I'_{j-1} \cup \{E_j\}$
- 10:  $C_j = C_{j-1} - 1$
- 11:  $\mathcal{M}_j = \frac{(N-j+1) \cdot \mathcal{M}_{j-1} - E_j}{N-j}$
- 12:  $\mathcal{D}_j = \frac{(N-j+1) \mathcal{D}_{j-1} - (E_j - \mathcal{M}_j) \cdot (E_j - \mathcal{M}_{j-1})}{N-j}$
- 13:  $SF_j = C_j \cdot (\mathcal{D}_0 - \mathcal{D}_j)$
- 14: **if**  $SF_j \geq SF(I_x)$  **then**
- 15:  $I_x = I'_j$
- 16:  $SF(I_x) = SF_j$
- 17: **end if**
- 18: **end for**
- 19: **return**  $I_x$

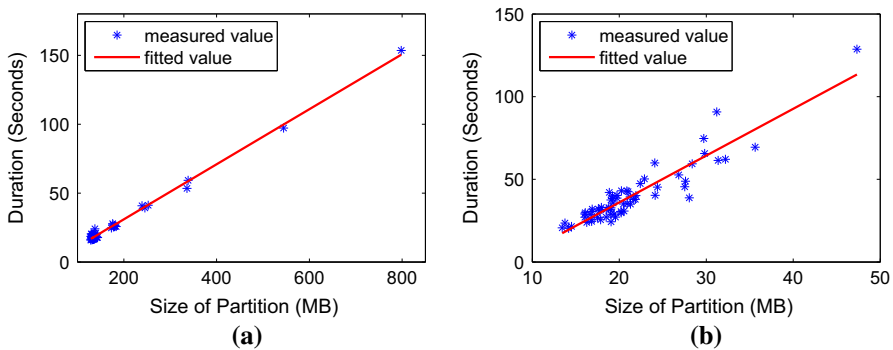
Observe that the bigger the distance from the mean, the larger the contribution it will induce on the variance. Thus, appending the elements one by one, in decreasing order of the distance from the mean, to the candidate exception set, the cardinality function of the residual set will continuously descend but the reduction in the variance will gradually ascend. As a result, at some point during the execution, we will observe the optimal subset where the smoothing factor is the greatest. Algorithm 1 presents the overloaded task detection algorithm in detail. First, we sort the elements in  $I$  in decreasing order of the distance from the mean.  $I'$  represents the sorted set, where  $I' = \{E_1, E_2, \dots, E_N\}$ . After that, we append the elements in  $I'$  one by one to the exception set  $I_x$ . Note that  $M_j$  is the mean of elements in the residual set of  $I'_j$ . More specially, according to Line 11 in Algorithm 1, when  $j = 1$ ,  $M_1 = \frac{(N \cdot M_0 - E_1)}{N-1}$ , which is the mean of elements in the residual set of  $I'_j$ , i.e.  $Mean(I' \setminus \{E_1\})$ . Similarly,  $D_j$  is the variance of elements in the residual set of  $I'_j$ .

**Theorem 2** *The time complexity of the overloaded tasks detecting algorithm is  $O(N \log(N))$ .*

*Proof* In line 1 of the Algorithm 1, sorting the set of  $N$  numbers requires  $O(N \log(N))$  computational complexity [26]. Subsequently, the operations from line 2–7 run in constant time  $O(1)$ . In the loop from line 8–17,  $M_j$  and  $D_j$  can be incrementally calculated by  $M_{j-1}$  and  $D_{j-1}$  respectively (see the derivation in [9]), and there are  $N$  iterations, thus, the time complexity of this loop is  $O(N)$ . The total complexity, consequently, is  $O(N \log(N))$ . □

### 4.3 Correlating Task Duration with Partition Size

Upon identifying the overloaded tasks, OPTIMA can now adjust the resource allocation to mitigate the partitioning skew. However, how much resources should be allocated to those tasks still needs to be decided. To achieve this, the impact of the partition size and resource allocation on the task duration should be determined. In this section, we present the relationship between task duration and partition size.



**Fig. 3** The relationship between task duration and partition size for Sort running on 10G synthetically skewed data, and InvertedIndex running on 10G Wikipedia data. **a** Sort. **b** InvertedIndex

We run a set of experiments in our testbed cluster (see details in Sect. 5.) and keep track of the task durations and partition sizes. More specifically, we fix the CPU and memory allocation of each reduce task and measure the task duration of each reduce task. The blue points in Figure 3 show the results of the partition size and corresponding task duration for every reduce tasks in the 10G Sort and InvertedIndex job. Then we use linear regression to determine this relationship with Eq. 3 as follow:

$$T_i = \alpha_2 + \beta_2 \cdot P_i, i \in [1, N] \tag{3}$$

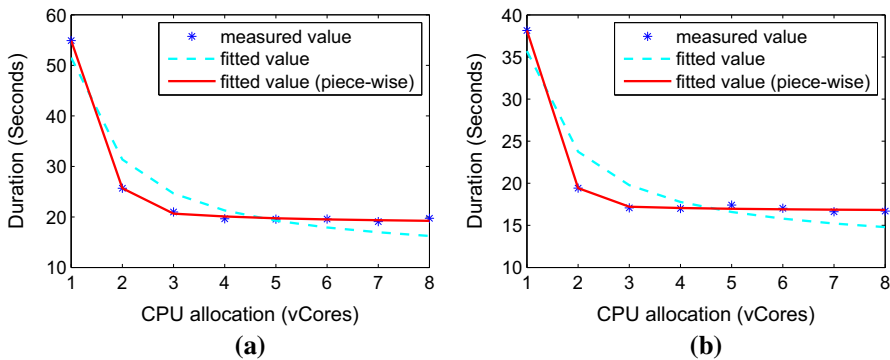
where  $T_i$  and  $P_i$  are the duration and partition size of reduce task  $i$  respectively. The results confirm that there is a linear relationship between task duration and partition size. Similar results have also been found by Lin et al. (see pp. 55–56 in [24]).

#### 4.4 Correlating Task Duration with Resource Allocation

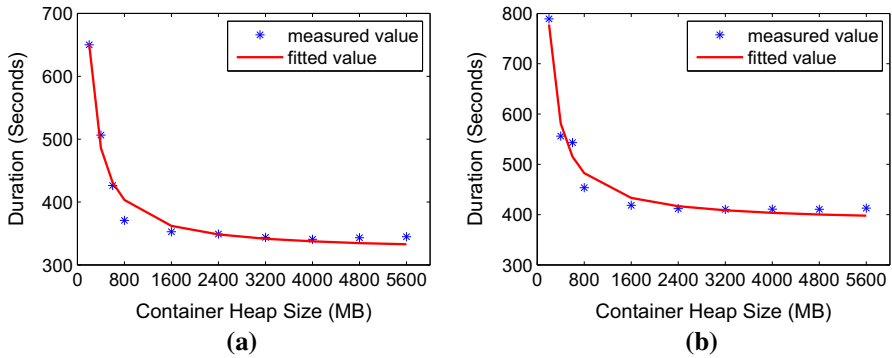
Similar to the previous section, we run a set of experiments by fixing the partition size and vary resource allocation of each reduce task to study the relationship between task duration and resource allocation. Figure 4 shows the task durations by varying the CPU allocation from 1 to 8 vCores (memory allocation is fixed to 1 GB) for 10G Sort and InvertedIndex job. We use non-linear regression [4] to model this relationship with following inversely proportional model:

$$T_i = \alpha_3 + \frac{\beta_3}{A_i^{cpu}}, i \in [1, N] \tag{4}$$

where  $A_i^{cpu}$  denotes the CPU allocation for reduce task  $i$ . The regression results is depicted by the dotted lines in Fig. 4. While this model fits well when the number of vCores is small, it is no longer accurate when a large amount of CPU resource is allocated to a task. This can be remedied by a piece-wise inversely proportional function. As shown in Fig. 4, the solid lines fit better than the dotted lines.



**Fig. 4** The relationship between task duration and CPU allocation for Sort running on 10G synthetically skewed data and Invertedindex running on 10G Wikipedia data. **a** Sort. **b** InvertedIndex



**Fig. 5** The relationship between task duration and memory allocation for Sort running on 30G synthetically skewed data and InvertedIndex running on 30G Wikipedia data. **a** Sort. **b** InvertedIndex

From above observations, it is clear that as the CPU allocation increases, the task duration reduces. However, after reaching a threshold (after 3 vCores in our experiments), the task duration does not decrease even though CPU allocation is continuously increasing. Similar observation is also made in Jalaparti et al. [18], where show that increasing the network bandwidth beyond a threshold does not help since the job completion time is dictated by disk performance. This is consistent with the phenomenon we observed.

In terms of memory, there are two configurations in YARN: *mapreduce.reduce.memory.mb* and *mapreduce.reduce.java.opts* (similar to map tasks), which are container RAM size and JVM heap size respectively. The former setting specifies the logical resource allocation of a task, which is used for headroom calculation in the ResourceManager and RAM usage monitoring in the NodeManager; the latter is the maximum heap size of the JVM process that executes the task, which specifies the actual memory size that a task can used. Therefore, we vary the JVM heap size setting from 200 (The default value) to 5600 MB while the CPU is fixed to 1 vCore, and model the relationship between task duration and memory allocation by following equation:

$$T_i = \alpha_4 + \frac{\beta_4}{A_i^{mem}}, i \in [1, N] \quad (5)$$

where  $A_i^{mem}$  is the memory allocation for reduce task  $i$ . Figure 5 shows the regression results for 30G Sort and 30G InvertedIndex respectively.<sup>1</sup> We can see that the inversely proportion model is also applicable to memory. More specifically, there is a significant improvement at the early portion of the curve. That is because the increase of memory can reduce disk I/O operations when memory requirements exceeds allocation. However, after a critical point, no improvement can be obtained even though memory allocation is increasing, which is consistent with the CPU resource.

<sup>1</sup> Since the memory requirement is related to the size of data, larger datasets are needed in order to clearly demonstrate the impact of memory allocation.

### 4.5 Resource Allocation Algorithm

We now present the resource allocation strategy in this section. The scheduler allocates resources for each reduce task in a way of minimizing the variation in their execution times, thus mitigating the impact of partitioning skew. From Sect. 4.3, we know that the task duration increases monotonically with the partition size. Hence, if each reduce task is allocated an identical amount of resource which is the solution of native YARN, those overloaded tasks will incur the variation in execution times among reduce tasks. On the other hand, we have demonstrated in Sect. 4.4 that the relationship between task duration and resource allocation follows an inverse proportional model. Therefore, we simply increase resource allocations for those overloaded tasks, keeping resource allocations for the ordinary tasks unchanged, thereby reducing the durations of overloaded tasks.

More specifically, in terms of memory, we allocate the amount that equals to their partition sizes and normalizes it with the memory allocation unit setting in YARN denoted as  $Unit^{mem}$ ; in terms of CPU, we scale up the allocation to  $\lfloor \frac{P_i}{P_{mean}} \rfloor \cdot A_{ord}^{cpu}$ , where  $P_{mean}$  is the mean of all the partitions and  $A_{ord}^{cpu}$  is the CPU allocation to ordinary tasks. This strategy is simple but works well in practice. It roughly estimates the resource requirements for overloaded tasks according to their partition sizes, and privileges these tasks by guaranteeing their resource allocation. Nevertheless, over-allocation will result in wasting resource as shown in Sect. 4.4. Besides, due to the finite resource capacities of nodes, resource allocations should be less than the capacities. We consider CPU and memory allocation should be less than threshold  $\varphi_{cpu}$  and  $\varphi_{mem}$  respectively, which are input parameters to our algorithm.

---

#### Algorithm 2 Resource allocation algorithm

---

**Input:**  $\delta$  - Threshold of stopping training the Partition Size Prediction Model;  
 $\varphi_{cpu}$  - Maximum allocation of CPU;  
 $\varphi_{mem}$  - Maximum allocation of memory;  
**Output:**  $C$  - Set of resource allocations for each reduce task  $\langle A_i^{cpu}, A_i^{mem} \rangle$

- 1: Collect  $S_i$  and  $F$ , when a success completion event of map tasks is received by ApplicationMaster;
- 2: When threshold  $\delta$  is reached:
- 3: Stop training and finalize Partition Size Prediction Model;
- 4: Predict the set  $\{P_i | i \in [1, N]\}$  while  $F = 100\%$ ;
- 5: Detect the overloaded tasks according to Algorithm 1, obtain the set  $I_x$ ;
- 6: Derive the set of task ID,  $ID_{skew}$ , from the detected outliers;
- 7: **for** each reduce task  $i \in ID_{skew}$  **do**
- 8:      $A_i^{mem} = Min \left( \lfloor \frac{P_i}{Unit^{mem}} \rfloor \cdot Unit^{mem}, \varphi_{mem} \right)$
- 9:     **if**  $i \in ID_{skew}$  **then**
- 10:          $A_i^{cpu} = Min \left( \lfloor \frac{P_i}{P_{mean}} \rfloor \cdot A_{ord}^{cpu}, \varphi_{cpu} \right)$
- 11:     **end if**
- 12:      $C = C \cup \{ \langle A_i^{cpu}, A_i^{mem} \rangle \}$ ;
- 13: **end for**
- 14: **return**  $C$

---

Algorithm 2 describes our resource allocation policy in detail. As shown in Line 1–4, after reaching the threshold  $\delta$ , the partition size of each reduce task can be predicted with the prediction model. Afterwards, the scheduler can detect the overloaded tasks and derive the corresponding task ID according to the estimated partition size. Finally, in the iteration from Line 7 to 13, the CPU allocation of overloaded tasks are adjusted according to their partition size. After the resource allocations are finalized, ApplicationMaster sends resource request to Fine-grained Container Scheduler in ResourceManager.

## 5 Evaluation

We have implemented OPTIMA on Hadoop YARN 2.4.0 as an additional feature. Implementing this approach requires only minimal change to the existing Hadoop architecture. And we perform our experiments on 11 virtual machines (VMs) in the SAVI Testbed [19], which contains a large cluster with many server machines. Each VM has four 2 GHz cores, 8 GB RAM and 80 GB hard disk. We deploy Hadoop YARN 2.4.0 with one VM as ResourceManager and NameNode, and remaining 10 VMs as workers. Each worker is configured with 8 virtual cores and 7 GB RAM (leaving 1 GB for other processes). The minimum CPU and memory allocations to a container are 1 vCore and 1 GB respectively. The HDFS block size is set to 64 MB, and the replication level is set to 3. The *CgroupsLCEResourcesHandler* configuration is enabled, and we also activate the configuration of map output compression.<sup>2</sup>

The applications used in our evaluation are as follows:

- 1 *Sort (SRT)* This application takes input data generated by RandomWriter as input, and outputs the data sorted by the key. Each map task sorts one split of the input dataset, and then each reduce task merges the output of the map tasks for a given partition key. Similar to [15], we modify RandomWriter to produce skewed data.
- 2 *InvertedIndex (II)* The inverted index is a popular data structure used for Web search, and this application takes a list of documents as input and generates an inverted index for these documents. Each map task emits  $\langle word, docId \rangle$  tuples and the reduce task combines all tuples on a assigned key and emits  $\langle word, list(docId) \rangle$  tuples.
- 3 *WordCount (WC)* WordCount computes the occurrence frequency of each word in the large collection of documents. Each map task emits  $\langle word, count \rangle$  pairs. The reduce task sums up the counts for a given key, which maybe several words, from all map tasks and outputs the final counts.
- 4 *RelativeFrequency (RF)* RelativeFrequency is introduced in [24]. Other than AbsoluteFrequency, which measures the number of times word  $w_i$  co-occurs with word  $w_j$  within a specific context, RF measures the proportion of time

<sup>2</sup> Using compression in Hadoop to optimize MapReduce performance is prevalent in industry and academia [6, 27, 34].



word  $w_j$  appears in the context of  $w_i$ . It is also denoted as  $F(w_j|w_i)$ . To compute  $F(w_j|w_i)$ . RF counts up the number of co-occurrences of the bigram  $(w_i, w_j)$ , and then divides it by the number of occurrences of all the bigrams  $(w_i, *)$ . We use the implementation of this application provided by [23].

Table 1 gives an overview of these benchmarks together with the configurations we have used in our experiments. The skewness of the reducer workload is measured by the coefficient of variation (CV), which is used as a fairness metric in the literature [17]. The larger the ratio, the more skewness is considered in the distribution of the partitions. We will present results of running these applications with both small and large datasets in the following sections.

### 5.1 Accuracy of Prediction of Partition Size

In this set of experiments, we wanted to validate the accuracy of the partition size prediction model. To this end, we execute MapReduce jobs on different datasets, and compute the mean absolute percentage error (MAPE) of all partitions in each scenario. The MAPE is defined as follows.

$$MAPE = \frac{1}{N} \sum_{i=1}^N \left| \frac{P_i^{pred} - P_i^{measrd}}{P_i^{measrd}} \right| \tag{6}$$

where  $N$  is the number of reduce tasks in this job,  $P_i^{pred}$  and  $P_i^{measrd}$  are the predicted and measured value of partition size of reduce task  $i$  respectively. Table 2 summarizes the MAPE for the testing applications with threshold  $\delta$  from 0.05 to 0.15 on the small and large datasets. We run 10 experiments for each scenario and adopt the average. It can be seen that the error rates of most of the applications are  $< 5\%$ . In particular, RelativeFrequency reaches the highest error at 7.84%. Furthermore, as the threshold  $\delta$  increased, the error rate of the prediction model decreases. This is not hard to understand, since more training data can be obtained as  $\delta$  increased, thereby improving the accuracy of the prediction model.

**Table 1** Benchmarks characteristics

Application	Dataset type	Input size (GB)	Skewness (%)	#map, reduce tasks	
Sort	RandomWriter	Small	5.086	31.52	85, 16
		Large	30.510	61.6	510, 64
InvertedIndex	Wikipedia	Small	5.759	16.94	92, 16
		Large	29.049	25.85	467, 64
WordCount	Wikipedia	Small	5.759	14.75	92, 16
		Large	29.049	23.47	467, 64
RelativeFrequency	Wikipedia	Small	5.759	62.62	92, 16
		Large	29.049	76.72	467, 64

## 5.2 Overloaded Task Detection Evaluation

We present next the results of applying the proposed overloaded task detection algorithm on real MapReduce workloads so as to evaluate its effectiveness.

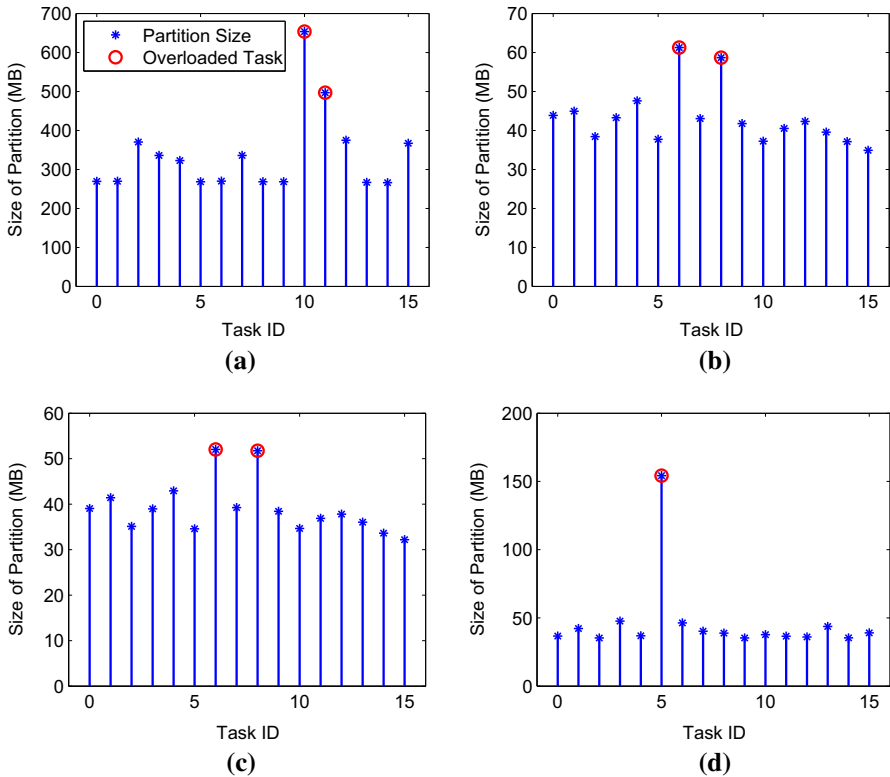
Figures 6 and 7 show the detection results by running our testing applications with Small dataset and Large dataset, respectively. We can see from these two figures that most of tasks have sizes hovering around the mean, and only several tasks have obviously larger sizes. The red circles identify the overloaded tasks detected by our detection algorithm. As the figures show, our algorithm accurately detect the overloaded tasks. In particular, those detected outliers of data skew are far from the mean than others in all cases except InvertedIndex on Large dataset, where some tasks that are slightly smaller than the most mildly outliers have escaped from the detection algorithm. That is because the algorithm tends to find the optimal subset that has the minimum number of the data skewed tasks but incurs greatest reduction on variance of the residual set. Even though those tasks are slightly smaller than the most mildly outliers, categorizing them to exception set can not reduce the variance significantly.

## 5.3 Performance Evaluation

In this section, we compare the performance of OPTIMA, DREAMS and native Hadoop YARN 2.4.0 (called Native in this paper). Note that tuning the number of reduce tasks of a MapReduce job can improve job completion time [39]. To isolate this effect, we use the same number of reduce tasks in the corresponding experiments when comparing the performance. Figure 8 shows the comparison among Native, DREAMS and OPTIMA with regard to the job completion time. We can see from the figure that OPTIMA outperforms Native for all cases. In particular, OPTIMA improves job completion time by 36.73 % for Sort on the 30GB dataset. Comparing to DREAMS, OPTIMA perform slightly inferior in some cases (e.g. WC), but overall, OPTIMA have equivalent gains.

**Table 2** Mean absolute percentage error of partition size prediction model on the small and large datasets

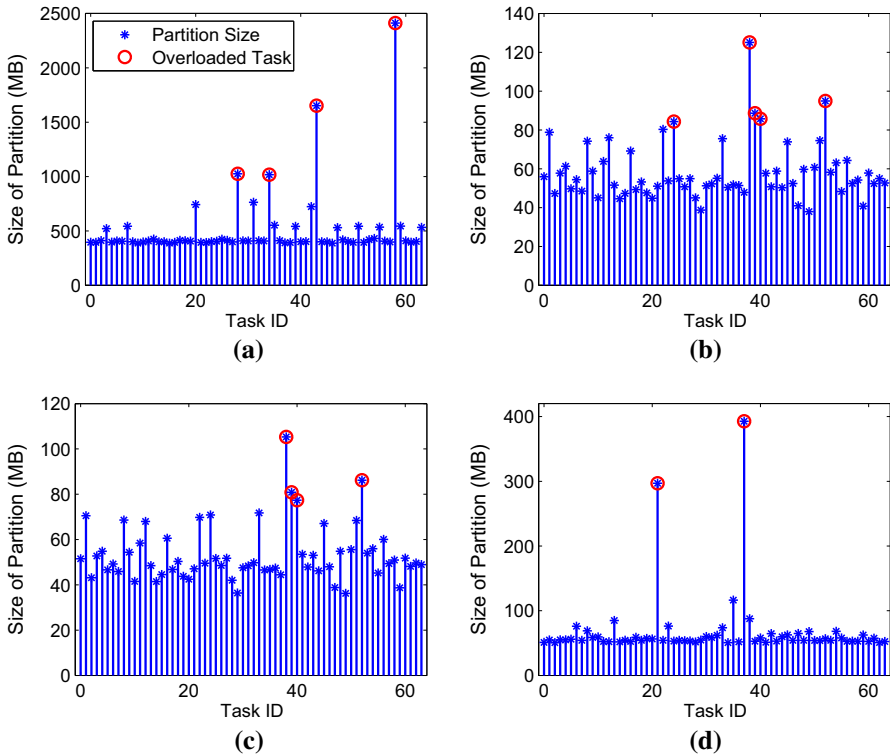
Application	Dataset	MAPE (%)			
		$\delta = 0.05$	$\delta = 0.08$	$\delta = 0.12$	
MAPE (%) $\delta = 0.15$					
Sort	Small	6.32	5.38	4.37	3.82
	Large	5.34	3.98	3.48	3.27
InvertedIndex	Small	3.69	2.86	2.49	2.37
	Large	3.40	2.46	1.81	1.57
WordCount	Small	5.34	3.98	3.48	3.27
	Large	3.94	2.8	2.07	1.85
RelativeFrequency	Small	7.84	6.04	4.63	4.12
	Large	6.99	5.29	4.31	4.08



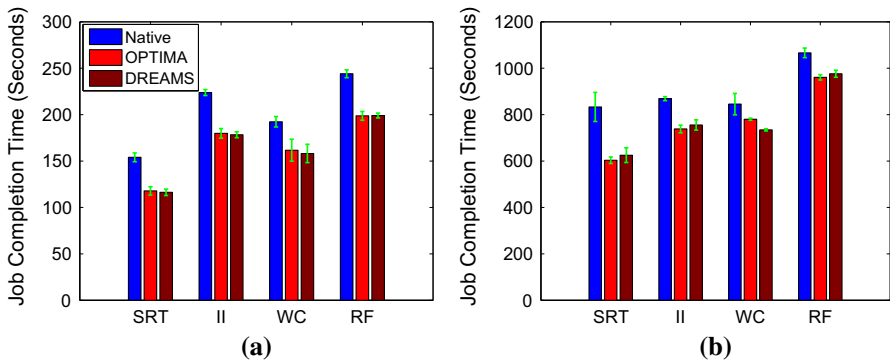
**Fig. 6** The results of the overloaded task detection on the small dataset. **a** Sort. **b** InvertedIndex. **c** WordCount. **d** RelativeFrequency

We also compare the makespan variance of reduce tasks in Native, DREAMS and OPTIMA. As we stated earlier, we try to eliminate the runtime difference among reduce tasks with different loads, thereby shortening the job completion time. Figure 9 shows the comparison results with respect to the CV of task durations for our benchmarks. The graphs illustrate that OPTIMA can effectively reduce the makespan variance of reduce tasks. More specially, the highest reduction ( $1.38 \times$  faster) is achieved when running RF on the 5 GB Wikipedia dataset. Since the shuffle phase in reduce stage is overlapping the entire map stage, it is not necessary to count the makespan when the reduce task is waiting for the output of map tasks. ARIA [33] takes only the non-overlapping portions of shuffle phase into account. Chowdhury et al. [7] also define the start of the shuffle phase as when either the last map task finishes or the last reduce task starts. Similarly, we compare the durations of reduce tasks starting from the last map task finishes.

We do not compare OPTIMA with any other existing data skew mitigating solutions because we cannot find an implementation of these solutions in YARN that support resource isolation. To the best of our knowledge, the existing solutions



**Fig. 7** The results of the overloaded task detection on the large dataset. **a** Sort. **b** InvertedIndex. **c** WordCount. **d** RelativeFrequency



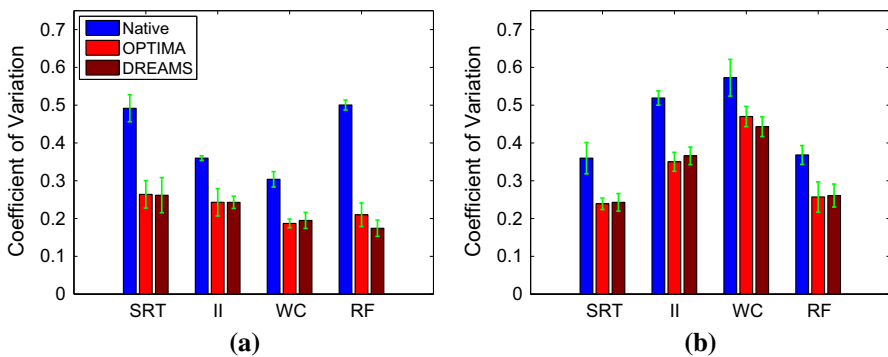
**Fig. 8** The comparison of job completion time between Native and OPTIMA. **a** Small dataset. **b** Large dataset

such as [16, 21, 29] are implemented on MRv1, which is slot-based and there is no isolation between slots. Therefore, in the evaluation, we only compare the OPTIMA with YARN 2.4.0 and our previous solution DREAMS.

## 6 Related Work

### 6.1 Handling Data Skew in MapReduce

The data skew problem in MapReduce has been extensively investigated recently. The authors, in [11] and [16], define a cost model for assigning reduce keys to reduce tasks so as to balance the load among reduce tasks. However, both approaches have to wait until all the map tasks have completed. As shown in [15], this would increase the job completion time. In order to equally distribute the load to worker machines while overlapping the map and reduce phase, the proposal in [22] applies a greedy balancing approach of assigning keys to the machine with the least load. This solution is based on the assumption that the size of each key-value pair is identical, which is not true in real workloads. Ramakrishnan et al. [29] propose a progressive sampler to estimate the intermediate data distribution and then partition the data to balance the load across all reduce tasks. However, this solution needs an additional sampling phase before jobs start, which can be time-consuming. Instead of chopping the large partitions to balance the load, SkewTune [21] repartitions heavily skewed partitions. However, it imposes an overhead while repartitioning data and concatenating original output. Finally, Zacheilas and Kalogeraki [37] propose DynamicShare, which aims at scheduling MapReduce jobs in heterogeneous systems to meet their real-time response requirements, and achieving an even distribution of the partitions by assigning the partitions in such a way that puts more work on powerful nodes. Similar to SkewTune, it imposes an overhead for the partition assignment procedure. Besides, DynamicShare cannot start partition assignment until all map tasks have completed. In an earlier work, we proposed DREAMS [25]. DREAMS dynamically allocates the right amount of resources to tasks to equalize the task completion time, which is simpler and incurs no overhead. However, DREAMS requires a job profiling stage, which limits its generality and applicability.



**Fig. 9** The comparison of makespan variance of reduce tasks between Native and OPTIMA. **a** Small dataset. **b** Large dataset

## 6.2 MapReduce Stragglers

Straggler problem has first been identified in [8], which backs up the execution of remaining in-progress tasks when the job is close to completion. LATE [38] extends this work by speculatively executing a replica task based on a simple heuristic of duplicating only those tasks at a slow progress rate. Because the replica task still has the same amount of the data to process, executing this speculative task may have counter-productive impact on resource utilization. Unlike [8, 38], Mantri [2] analyzes the causes of stragglers in MapReduce clusters and culls stragglers based on their causes. With respect to data skew, Mantri schedules tasks in descending order of their input size to mitigate skew. However, Mantri assumes the sizes of input data are known before a stage starts, which is not the case in MapReduce framework. And scheduling tasks in descending order of input sizes are complementary to our work. Wrangler [36] predicts whether the worker nodes will create a straggler based on their runtime resource usage statistics. If a node is predicted to create a straggler, Wrangler will not assign tasks to it. As a result, it can proactively avoid overloading of nodes. However, Wrangler neglects that the straggling situation can also be incurred by the task itself; partitioning skew is one such example. The tasks with extremely large partitions may still lead to the straggling situation. Wrangler considers each task has unique resource requirement, but it is not true when partitioning skew exists. For reducers with large partitions, even though Wrangler will assign them to nodes that behaved normally, these reducers still need more time to process than reducers with smaller partitions. And therefore, it will not change much in this case. By contrast, OPTIMA proactively adjusts resource allocation to the reducers with large partition, which guarantees the resource for these tasks with the goal of reducing the variation of task durations, and hence accelerating the job completion.

## 6.3 Resource-Aware Scheduling

Resource-aware scheduling has received considerable attention in recent years. The original Hadoop MapReduce (*i.e.* MRv1) implements a slot-based resource allocation scheme, which does not take run-time task resource consumption into consideration. To address this limitation, Hadoop YARN [32] represents a major endeavor towards resource-aware scheduling in MapReduce clusters. It offers the ability to specify the size of container in terms of requirements for each type of resources. However, YARN assumes the resource consumption for each Map (or Reduce) task in a job is identical, which is not true for data skewed MapReduce jobs. Sharma et al. [30] propose MROrchestrator, a MapReduce resource framework that can identify the resource deficit based on resource profiling, and dynamically adjusts resource allocation. Compared to our solution, MROrchestrator cannot proactively identify stragglers due to workload imbalance before task launch. In other words, if all overloaded tasks are launched in a machine, no matter how the MROrchestrator adjusts the allocation, resource deficit cannot be mitigated. There are several other proposals that fall in another category of resource scheduling policies such as [10, 28, 33, 35]. The main focus of these approaches is on adjusting

the resource allocation in terms of the number of map and reduce slots for the jobs in order to achieve fairness, maximize resource utilization or meet job deadline. These however do not address the data skew problem.

## 7 Conclusion

In this paper, we present OPTIMA, an on-line partitioning skew mitigation technique for MapReduce with resource adjustment. Rather than gathering the workload statistics until all the map tasks have completed, or sampling them before jobs run, OPTIMA predicts the workload distribution of reduce tasks on-line, which does not incur synchronization barriers. Further, based on the predicted workload distribution, OPTIMA detects the overloaded tasks using a deviation detection technique in linearithmic time. And finally, OPTIMA pro-actively adjusts resource allocation to the overloaded tasks. This can eliminate the makespan difference among reduce tasks, thereby accelerating the job completion. Compared with our previous work DREAMS, OPTIMA abandons the profiles and eliminates the dependency on the task performance model. This not only eliminates the limitation in terms of applicability of DREAMS, but also allows the solution to be carried out in an on-line manner. In our experiments using an 11-node cluster running both real and synthetic workloads, we show that our on-line partition size prediction algorithm achieve high accuracy with 7.84 % relative error in the worst case. We then demonstrate that OPTIMA can effectively mitigate the negative impact of partitioning skew, thereby improving the job running time by up to 36.73 % compared to the native Hadoop in our experiments. While compared to DREAMS, OPTIMA can deliver the same gain as DREAMS in a much more efficient manner.

**Acknowledgments** This work is supported in part by the National Natural Science Foundation of China (No. 61472438), and in part by the Smart Applications on Virtual Infrastructure (SAVI) project funded under the National Sciences and Engineering Research Council of Canada (NSERC) Strategic Networks Grant Number NETGP394424-10.

## References

1. Ananthanarayanan, G., Hung, M.C.C., Ren, X., Stoica, I., Wierman, A., Yu, M.: Grass: trimming stragglers in approximation analytics. In: Proceedings of the 11th USENIX NSDI (2014)
2. Ananthanarayanan, G., Kandula, S., Greenberg, A.G., Stoica, I., Lu, Y., Saha, B., Harris, E.: Reining in the outliers in map-reduce clusters using mantri. In: OSDI, vol. 10, p. 24. (2010)
3. Arning, A., Agrawal, R., Raghavan, P.: A linear method for deviation detection in large databases. In: KDD, pp. 164–169. (1996)
4. Bates, D.M., Watts, D.G.: Nonlinear Regression: Iterative Estimation and Linear Approximations. Wiley, New Jersey (1988)
5. Borthakur, D.: The hadoop distributed file system: architecture and design. Hadoop Proj. Website **11**, 21 (2007)
6. Chen, Y., Ganapathi, A., Katz, R.H.: To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency. In: Proceedings of the First ACM SIGCOMM Workshop on Green Networking, pp. 23–28. ACM (2010)

7. Chowdhury, M., Zaharia, M., Ma, J., Jordan, M.I., Stoica, I.: Managing data transfers in computer clusters with orchestra. In: ACM SIGCOMM Computer Communication Review, vol. 41, pp. 98–109. ACM (2011)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. *Commun. ACM* **51**(1), 107–113 (2008)
9. Finch, T.: Incremental calculation of weighted mean and variance. University of Cambridge, Cambridge (2009)
10. Ghodsi, A., Zaharia, M., Hindman, B., Konwinski, A., Shenker, S., Stoica, I.: Dominant resource fairness: fair allocation of multiple resource types. In: NSDI, vol. 11, pp. 24–24 (2011)
11. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Handling data skew in mapreduce. In: Proceedings of the 1st International Conference on Cloud Computing and Services Science, vol. 146, pp. 574–583 (2011)
12. Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Load balancing in mapreduce based on scalable cardinality estimates. In: Data Engineering (ICDE), 2012 IEEE 28th International Conference on, pp. 522–533. IEEE (2012)
13. Hadoop mapreduce distribution <http://hadoop.apache.org/docs/r1.2.1/>
14. Hadoop: Fair scheduler <http://hadoop.apache.org/docs/r2.4.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html>
15. Hammoud, M., Rehman, M.S., Sakr, M.F.: Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In: Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on, pp. 49–58. IEEE (2012)
16. Ibrahim, S., Jin, H., Lu, L., He, B., Antoniu, G., Wu, S.: Handling partitioning skew in mapreduce using leen. *Peer Peer Netw. Appl.* **6**(4), 409–424 (2013)
17. Jain, R., Chiu, D.M., Hawe, W.R.: A quantitative measure of fairness and discrimination for resource allocation in shared computer system (1984)
18. Jalaparti, V., Ballani, H., Costa, P., Karagiannis, T., Rowstron, A.: Bridging the tenant-provider gap in cloud services. In: Proceedings of the Third ACM Symposium on Cloud Computing, p. 10. ACM (2012)
19. Kang, J.M., Bannazadeh, H., Leon-Garcia, A.: Savi testbed: Control and management of converged virtual ict resources. In: IFIP/IEEE International Symposium on Integrated Network Management (IM 2013), 2013 pp. 664–667. IEEE (2013)
20. Kirby, G.: Zipf’s law. *UK J. Nav. Sci.* **10**(3), 180–185 (1985)
21. Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skewtune: mitigating skew in mapreduce applications. In: Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, pp. 25–36. ACM (2012)
22. Le, Y., Liu, J., Ergun, F., Wang, D.: Online load balancing for mapreduce with skewed data input. In: INFOCOM, 2014 Proceedings IEEE, pp. 2004–2012. IEEE (2014)
23. Lin, J.: Cloud 9: A mapreduce library for hadoop (2010a). <https://github.com/lintool/Cloud9>
24. Lin, J., Dyer, C.: Data-intensive text processing with mapreduce. *Synth. Lect. Hum. Lang. Technol.* **3**(1), 1–177 (2010b)
25. Liu, Z., Zhang, Q., Zhani, M.F., Boutaba, R., Liu, Y., Gong, Z.: Dreams: Dynamic resource allocation for mapreduce with data skew. In: IFIP/IEEE International Symposium on Integrated Network Management (IM 2015), 2015. Ottawa (2015)
26. Papadimitriou, C.H.: Computational Complexity. Wiley, New Jersey (2003)
27. Papers, M.I.W.: Compression in hadoop. <http://technet.microsoft.com/en-us/library/dn247618.aspx>
28. Polo, J., Carrera, D., Becerra, Y., Torres, J., Ayguadé, E., Steinder, M., Whalley, I.: Performance-driven task co-scheduling for mapreduce environments. In: Network Operations and Management Symposium (NOMS), 2010 IEEE, pp. 373–380. IEEE (2010)
29. Ramakrishnan, S.R., Swart, G., Urmanov, A.: Balancing reducer skew in mapreduce workloads using progressive sampling. In: Proceedings of the Third ACM Symposium on Cloud Computing, p. 16. ACM (2012)
30. Sharma, B., Prabhakar, R., Lim, S., Kandemir, M.T., Das, C.R.: Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters. In: IEEE 5th International Conference on Cloud Computing (CLOUD), 2012, pp. 1–8. IEEE (2012)
31. Tan, P.N., Steinbach, M., Kumar, V., et al.: Introduction to Data Mining, vol. 1. Pearson Addison Wesley, Boston (2006)



32. Vavilapalli, V.K., Murthy, A.C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., et al.: Apache hadoop yarn: Yet another resource negotiator. In: Proceedings of the 4th annual Symposium on Cloud Computing, p. 5. ACM (2013)
33. Verma, A., Cherkasova, L., Campbell, R.H.: Aria: automatic resource inference and allocation for mapreduce environments. In: Proceedings of the 8th ACM International Conference on Autonomic Computing, pp. 235–244. ACM (2011)
34. White, T.: Hadoop: The definitive guide. O'Reilly Media Inc, California (2012)
35. Wolf, J., Rajan, D., Hildrum, K., Khandekar, R., Kumar, V., Parekh, S., Wu, K.L., Balmin, A.: Flex: A slot allocation scheduling optimizer for mapreduce workloads. In: Middleware 2010, pp. 1–20. Springer (2010)
36. Yadwadkar, N.J., Ananthanarayanan, G., Katz, R.: Wrangler: Predictable and faster jobs using fewer resources. In: Proceedings of the ACM Symposium on Cloud Computing, pp. 1–14. ACM (2014)
37. Zacheilas, N., Kalogeraki, V.: Real-time scheduling of skewed mapreduce jobs in heterogeneous environments. In: Proceedings of 11th International Conference on Autonomic Computing, pp. 189–200. USENIX (2014)
38. Zaharia, M., Konwinski, A., Joseph, A.D., Katz, R.H., Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: OSDI, vol. 8, p. 7 (2008)
39. Zhang, Z., Cherkasova, L., Loo, B.T.: Autotune: Optimizing execution concurrency and resource usage in mapreduce workflows. In: ICAC, pp. 175–181 (2013)

**Zhihong Liu** received the B.A.Sc. and M.Sc. degrees in computer science from South China University of Technology and National University of Defense Technology, respectively. He is a Ph.D. candidate in National University of Defense Technology with research interests in big-data analytics and resource management in cloud computing. Currently, he is a visiting student in University of Waterloo, Canada.

**Qi Zhang** received the B.A.Sc., M.Sc. and Ph.D. from University of Ottawa (Canada), Queen's University (Canada) and University of Waterloo (Canada), respectively. His current research focuses on resource management for cloud computing systems. He is currently pursuing a Post-doctoral fellowship at University of Toronto (Canada) He is also interested in related areas including big-data analytics, software-defined networking, network virtualization and management.

**Raouf Boutaba** received the M.Sc. and Ph.D. degrees in computer science from the University Pierre and Marie Curie, Paris, France, in 1990 and 1994, respectively. He is currently a Professor of computer science with the University of Waterloo, Waterloo, ON, Canada. His research interests include control and management of networks and distributed systems. He is a fellow of the IEEE and the Engineering Institute of Canada.

**Yaping Liu** received the Ph.D. degree in computer science from National University of Defense Technology, China, in 2006. She is currently a Professor in College of Computer with National University of Defense Technology. Her current research interests include network architecture, inter-domain routing, network virtualization and network security.

**Baosheng Wang** received the B.A.Sc., M.Sc., and Ph.D. degrees in computer science from National University of Defense Technology, China, in 1992, 1995 and 2005, respectively. He is a Professor in College of Computer with National University of Defense Technology. His research interests are network management, distributed systems and network virtualization.