Dynamic Resource Allocation for MapReduce with Partitioning Skew

Zhihong Liu, *Student Member, IEEE*, Qi Zhang, *Student Member, IEEE*, Reaz Ahmed, *Member, IEEE*, Raouf Boutaba, *Fellow, IEEE*, Yaping Liu, and Zhenghu Gong

Abstract—MapReduce has become a prevalent programming model for building data processing applications in the cloud. While being widely used, existing MapReduce schedulers still suffer from an issue known as partitioning skew, where the output of map tasks is unevenly distributed among reduce tasks. Existing solutions follow a similar principle that repartitions workload among reduce tasks. However, those approaches often incur high performance overhead due to the partition size prediction and repartitioning. In this paper, we present DREAMS, a framework that provides run-time partitioning skew mitigation. Instead of repartitioning workload among reduce tasks, we cope with the partitioning skew problem by controlling the amount of resources allocated to each reduce task. Our approach completely eliminates the repartitioning overhead, yet is simple to implement. Experiments using both real and synthetic workloads running on a 21-node Hadoop cluster demonstrate that DREAMS can effectively mitigate the negative impact of partitioning skew, thereby improving the job completion time by up to a factor of 2.29 over the native Hadoop YARN. Compared to the state-of-the-art solution, DREAMS can improve the job completion time by a factor of 1.65.

Index Terms—MapReduce, Hadoop YARN, resource allocation, partitioning skew

1 Introduction

In recent years, the exponential growth of raw data has generated tremendous needs for large-scale data processing. In this context, MapReduce [1], a parallel computing framework, gained significant popularity. A MapReduce job consists of two types of tasks, namely *Map* and *Reduce*. Each map task takes a chunk of input data and runs a user-specified map function to generate intermediate key-value pairs. Subsequently, each reduce task collects the intermediate key-value pairs and applies a user-specified reduce function to produce the final output. Due to its remarkable advantages in terms of simplicity, robustness and scalability, MapReduce has been widely used by companies such as Amazon, Facebook, and Yahoo! to process large volumes of data on a daily basis. Consequently, it has attracted considerable attention from both industry and academia.

Despite its success, the current implementations of MapReduce suffer from a few limitations. In particular, the widely-used MapReduce system, Apache Hadoop MapReduce [2], uses a hash function to partition the intermediate key-value pairs across reduce tasks. The goal of using a hash function is to evenly distribute the workload to each reduce task. In reality this goal is rarely achieved [3], [4]. For example, Zacheilas *et al.* [3] have demonstrated the existence of skewness in a Youtube social graph application using real-world data. The experiments in [3] showed that the biggest workload among reduce tasks is larger than the smallest by more than a factor of five. ***The skewed workload distribution among reduce tasks can have a severe impact on job completion time. Note that the completion time of a MapReduce job is determined by the completion time of the slowest reduce task. Data skewness causes certain tasks with heavy workload run slower than others. This in turn prolongs the job completion time.***

Several recent approaches are proposed to handle the partitioning skew problem [4], [5], [6], [7], [8], [9], [10]. They follow a similar principle that predicts the workload for individual reduce tasks based on certain statistics of key-value pairs (*e.g.* key frequencies [6], [8]), and then repartitions the workload to achieve a better balance among the reduce tasks. However, in order to collect the statistics of key-value pairs, most of those solutions either have to prevent the reduce phase from overlapping with the map phase, or add a sampling phase before executing the actual job. Skewtune [4] can reduce this waiting time by redistributing the unprocessed workload of a slow reduce task at run-time. However, Skewtune incurs an additional run-time overhead of approximately 30 seconds (as reported in [4]). This overhead can be quite expensive for small jobs with average life span of around 100 seconds, which are very common in today's production clusters [11].

Motivated by the limitations of the existing solutions, in this paper, we take a radically different approach to address data skewness. Instead of repartitioning the workload among reduce tasks, our approach dynamically allocates resources to reduce tasks according to their workload. Since no repartitioning is involved, our approach completely eliminates the repartitioning overhead. To this end, we present DREAMS, a **D**ynamic **RE**source **A**llocation technique for **M**apReduce with partitioning **S**kew. DREAMS leverages historical records to construct profiles for each job type. This is reasonable because many production jobs are executed

- *Zhihong Liu is with the College of Computer, National University of Defense Technology, Changsha, China and David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON, Canada. E-mail: zhliu@nudt.edu.cn*
- *Q. Zhang, R. Ahmed and R. Boutaba are with David R. Cheriton School of Computer Science, University of Waterloo. Yaping Liu and Zhenghu Gong are with National University of Defense Technology*

Manuscript received April 19, 2005; revised September 17, 2014.

repeatedly in today's production clusters [12]. At run-time, DREAMS can dynamically detect data skewness and assign more resources to reduce tasks with large partitions to make them finish faster. Compared to the previous works, our contributions can be summarized as follows:

- We first develop a partition size prediction model that can forecast the partition sizes of reduce tasks at run-time. Specifically, we can accurately predict the size of each partition when only 5% of map tasks have completed.
- We establish a task performance model that correlates the completion time of individual reduce tasks with their partition sizes and resource allocation.
- We propose a scheduling algorithm that dynamically adjusts resource allocation to each reduce task using our task performance model and the estimation of the partition size. This can reduce the running time difference among reduce tasks that have different sizes of partitions to process, thereby accelerating the job completion.

Experiments using both real and synthetic workloads running on a 21-node Hadoop cluster demonstrate that DREAMS can effectively mitigate the negative impact of partitioning skew, thereby improving the job completion time by up to a factor of 2.29 over the native Hadoop YARN. Compared to the state-of-the-art solution like SkewTune, DREAMS can improve the job completion time by a factor of 1.65.

This paper extends our preliminary work [13] in a number of ways. First, the time complexity of the on-line partition size prediction model has been presented. Second, we have added memory allocation into the reduce task performance model. Third, the scheduling algorithm in the original manuscript has been reformulated as an optimization problem and its optimal solution is presented. Finally, we have conducted additional experiments to evaluate the effectiveness of DREAMS.

The rest of this paper is organized as follows. Section 2 provides the motivations of our work. We describe the system architecture of DREAMS in Section 3. Section 4 illustrates the design of DREAMS in detail. Section 5 provides the results from experimental evaluation. Finally, we summarize the existing works related to DREAMS in Section 7, and draw our conclusion in Section 8.

2 MOTIVATION

In the state-of-the-art MapReduce systems, each map task processes one chunk of the input data, and generates a sequence of intermediate key-value pairs. A hash function is then used to partition these key-value pairs and distribute them to reduce tasks. Since all map tasks use the same hash function, the key-value pairs with the same hash value are assigned to the same reduce task. During the reduce stage, each reduce task takes one partition (i.e. the intermediate key-value pairs corresponding to the same hash value) as input, and performs a user-specified reduce function on its partition to generate the final output. This process is illustrated in Figure 1. Ideally, the hash function is expected to generate equal size partitions if the key frequencies,

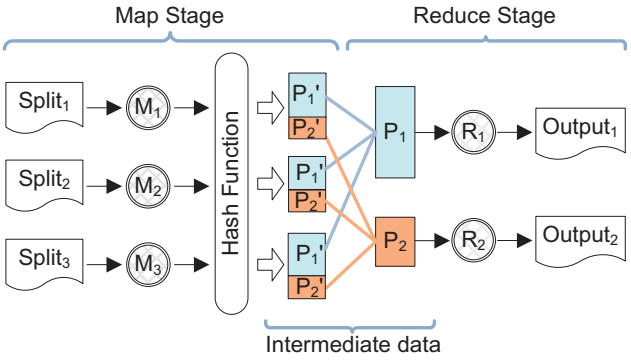


Fig. 1: MapReduce Programming Model

and sizes of the key-value pairs are uniformly distributed. However, in reality, the hash function often fails to achieve uniform partitioning, resulting into skewed partition sizes. For example in the InvertedIndex job [14], the hash function partitions the intermediate key-value pairs based on the occurrence of words in the files. Therefore, reduce tasks processing more popular words will be assigned a larger number of key-value pairs. As shown in Figure 1, partitions are unevenly distributed by the hash function. P_1 is larger than P_2, which causes workload imbalance between R_1 and R_2. Zacheilas *et al.* [3] presented the following reasons of partitioning skew:

- *Skewed key frequencies*: Some keys occur more frequently in the intermediate data. As a result, partitions that contain these keys become extremely large, thereby overloading the reduce tasks that they are assigned to.
- *Skewed tuple sizes*: In MapReduce jobs where sizes of the values in the key-value pairs vary significantly, even though key frequencies are uniform, uneven workload distribution among reduce tasks may arise.

In order to address the weaknesses and inadequacies experienced in the first version of Hadoop MapReduce (MRv1), the next generation of the Hadoop compute platform, YARN [15], has been developed. Compared to MRv1, YARN manages the scheduling process using two components: a) *ResourceManager* is responsible for allocating resources to the running MapReduce jobs subject to capacity constraints, fairness and so on; b) an *ApplicationMaster*, on the other hand, works for each running job, and has the responsibility of negotiating appropriate resources from *ResourceManager* and assigning the obtained resources to its tasks. This removes the single point bottleneck of JobTracker in MRv1 and improves the ability to scale Hadoop clusters. In addition, YARN deprecates the slot-based resource management approach in MRv1, and adopts a more flexible resource unit called *container*. The container provides resource-specific, fine-grain accounting (e.g. $< 2\,GB\,RAM, 1\,CPU >$). A task running within a container is enforced to abide by the prescribed limits.

Nevertheless, in both Hadoop MRv1 and YARN, the schedulers assume each reduce task has uniform workload and resource consumption, and therefore allocate identical resources to each reduce task. Specifically, MRv1 adopts a

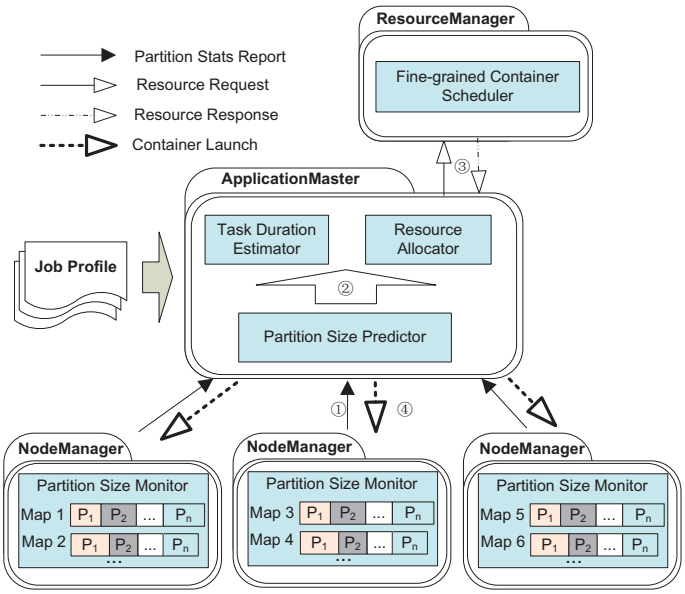


Fig. 2: Architecture of DREAMS

slot-based allocation scheme, where each machine is divided into identical "slots" that can be used to execute tasks. However, MRv1 does not provide resource isolation among co-located tasks, which may cause performance degradation at run-time [16]. On the other hand, YARN uses a container-based allocation scheme, where each task is scheduled in an isolated container. But, it still allocates containers of identical size to all reduce tasks that belong to the same job. ***This scheduling scheme can cause variation in task running time due to partitioning skew, since the execution time of a reduce task with a large partition can be prolonged because of the fixed container size. As the job completion time is dominated by the slowest task, the run-time variation of reduce tasks will prolong the job execution time.***

Most of the existing approaches tackle the partitioning skew problem by making the workload assignment uniformly distributed among reduce tasks, thereby mitigating the inefficiencies in both performance and utilization. However, achieving this goal requires (sometimes heavy) modification to the current Hadoop implementation, and often requires additional overhead in terms of sampling and adaptive partitioning. Therefore, in this work we seek an alternative solution, where we adjust the size of the container based on partitioning skew. This approach not only requires minimal modification to the existing Hadoop implementation, but at the same time effectively mitigates the negative impact of partitioning skew.

3 System Architecture

This section describes the design of our proposed resource allocation framework called DREAMS. The architecture of DREAMS is shown in Figure 2. There are five main components: *Partition Size Monitor*, running in the NodeManager; *Partition Size Predictor*, *Task Duration Estimator* and *Resource Allocator*, running in the ApplicationMaster; and *Fine-grained Container Scheduler*, running in the ResourceManager. Each *Partition Size Monitor* records the statistics of intermediate data that a map task generates at run-time and sends them to the ApplicationMaster though heartbeat messages. The *Partition Size Predictor* collects the partition size reports from NodeManagers and predicts the partition sizes of every reduce task for this job. The *Task Duration Estimator* constructs statistical estimation model of reduce task performance as a function of its partition size and resource allocation. That is, the duration of a reduce task can be estimated if the partition size and resource allocation of this task are given. The *Resource Allocator* determines the amount of resources to be allocated to each reduce task based on the performance estimation. Lastly, the *Fine-grained Container Scheduler* is responsible for scheduling resources among all the ApplicationMasters in the cluster, based on scheduling policies such as Fair scheduling [17] and Dominant Resource Fairness (DRF) [18]. Note that the schedulers in original Hadoop assume that all reduce tasks (and similarly, all map tasks) have homogeneous resource requirements in terms of CPU and memory. However, this is not appropriate for MapReduce jobs with partitioning skew. We have modified the original schedulers to support fine-grained container scheduling that allows each task to request resources of customizable size.

The workflow of resource allocation mechanism used by DREAMS consists of 4 steps as shown in Figure 2.

(1) After the ApplicationMaster is launched, it schedules all the map tasks first and then ramps up the reduce task requests gradually according to the *slowstart* setting, which is used to control when to start reduce tasks based on the percentage of map tasks that have finished. During their execution, each *Partition Size Monitor* records the size of intermediate key-value pairs produced by map tasks. Each *Partition Size Monitor* sends locally gathered statistics to the ApplicationMaster through the *TaskUmbilicalProtocal*, which is a RPC protocol used to monitor task status in Hadoop.

(2) Upon receiving the partition size reports from the *Partition Size Monitors*, the *Partition Size Predictor* performs size prediction using our proposed prediction model (see Section 4.1). After all the estimated sizes of reduce tasks are known, the *Task Duration Estimator* uses the reduce task performance model (Section 4.2) to predict the duration of each reduce task with specified amount of resources. Based on that, the *Resource Allocator* determines the amount of resources for each reduce task according to our proposed resource allocation algorithm (Section 4.3) to equalize the execution time of all reduce tasks and then sends resource requests to the ResourceManager. ***Note that the ResourceManager reports to the ApplicationMaster the current total amount of available resources through heartbeat messages every second. Thus, the Resource Allocator can check the availability of resources when requesting containers.***

(3) Next, the ResourceManager receives ApplicationMasters' resource requests through the heartbeat messages, and schedules free containers in the cluster to corresponding ApplicationMasters.

(4) Once the ApplicationMaster obtains new containers from the ResourceManager, it assigns the corresponding containers to the pending tasks, and finally launches the tasks.

4 DREAMS DESIGN

There are three main challenges to be addressed in DREAMS. First, in order to identify partitioning skew, it is necessary to develop a run-time forecasting algorithm that predicts the partition size of each reduce task. Second, in order to determine the right container size for each reduce task, it is necessary to develop a task performance model that correlates task running time with resource allocation. Lastly, there are multiple resource dimensions such as CPU, memory, disk I/O and network bandwidth. Allocations with different combination of these resource dimensions may yield the same completion time. Determining the appropriate combination of these resource dimensions in order to minimize the cost is a challenging problem. In the rest of this section, we shall describe our solutions for each of these challenges.

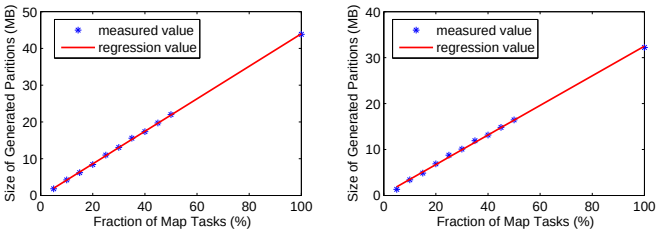
4.1 Predicting Partition Size

In order to cull the partitioning skew, the workload distribution among the reduce tasks should be known in advance. Unfortunately, the size of the partition belonging to each reduce task really depends on the input dataset, the map function and the number of reduce tasks in a MapReduce job. Even though most of the MapReduce jobs are routinely executed, the same job processing different input dataset would produce different workload distribution among its reduce tasks. Several recently proposed approaches calculate the workload distribution among reduce tasks [3], [5], [6], [7], [19]. Existing solutions, however, either have to wait for all the map tasks to finish [3], [5], [6], or need an additional sampling procedure before executing a job [7], [19]. However, in order to improve the job completion time, existing Hadoop schedulers allow reduce tasks to be launched before the completion of all map tasks (e.g. the default *slowstart* setting is 5%). It has also been demonstrated by the existing works [8], [20] that starting the shuffle phase after the completion of all the map tasks will severely prolong the job completion time. Therefore, it is necessary to predict the partition size at run-time without introducing a barrier between map and reduce phases.

The input datasets of MapReduce jobs in a production cluster tend to be very large. Hence, the HDFS storage system [21] splits a large dataset into smaller data chunks, which naturally creates a sampling space. This suggests that a small set of random samples in this sample space may reveal the characteristics of the whole dataset in terms of workload distribution among reduce tasks. Therefore, we can analyze the pattern of the intermediate data after a fraction of map tasks have completed, and then predict workload distribution among reduce tasks for the entire dataset.

In DREAMS, we perform k measurements ($j = 1, 2, ..., k$) over time during the map phase, and collect the following two metrics $\left(F^j,\ S_i^j\right)$:

- F^j is the percentage of map tasks that have been processed, where $j \in [1, k]$ and k refers to the number of collected tuples $\left(F^j,\ S_i^j\right)$. Note that each map task processes one inputsplit, and each inputsplit has identical size (64MB, 128MB etc.). As a result, F^j is approximately equal to the fraction of whole dataset that has been processed.
- 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 completions.



(a) a reduce task in InvertedIndex (b) a reduce task in WordCount

Fig. 3: Partition size prediction

Our experimental evidences reveal that S_i^j is linearly proportional to F^j. Figure 3 shows the typical results in InvertedIndex and WordCount jobs. Note that when 100% map tasks are completed, S_i^j will represent the actual partition size for reduce task i. Hence, we use linear regression to determine the following equation for each reduce task $i \in [1, N]$:

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

where α_1 and β_1 are the regression coefficients. We introduce an outer factor, δ, which works as a threshold to control our prediction model to stop the training process, and finalize the prediction. In practice, δ can be the map completion percentage (e.g. 5%) at which scheduling of the reduce tasks may be started. Every time a new map task has finished, a new training data is generated. When the fraction of map tasks reaches δ, we calculate the regression coefficients (α_1, β_1), and predict the partition size for each reduce task. ***Note that k is determined by δ. For instance, consider there are 100 map tasks in the job, if δ = 5%, then k = 5.***

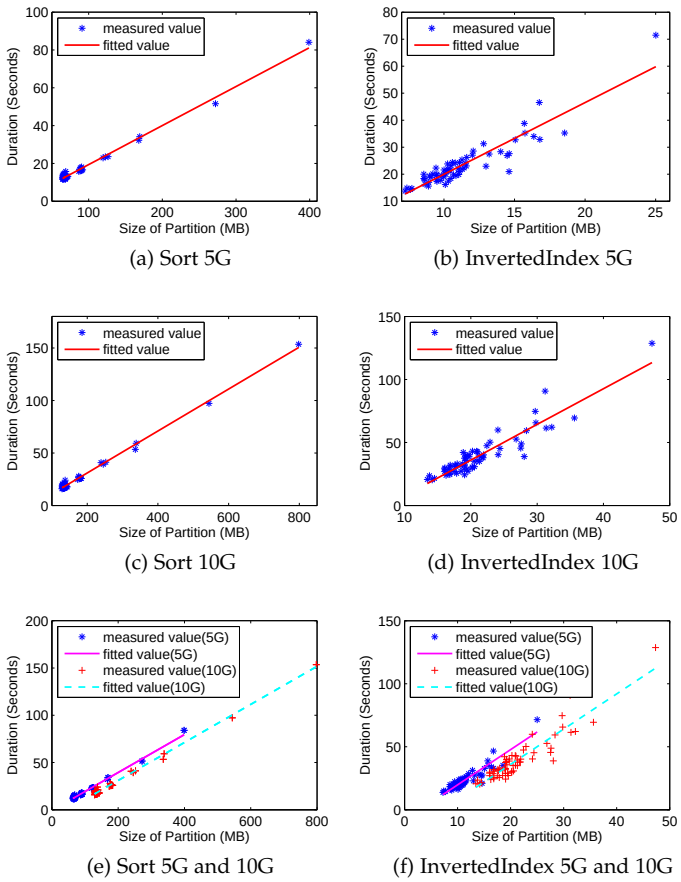
The computational complexity of our on-line partition size prediction model is $O(k \cdot N)$. In particular, for each reduce task $i \in [1, N]$, the scaling factors can be determined by the following equation:

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

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 $O(2^2 k)$ to multiply X^T by X, $O(2^3)$ to compute the inverse of $X^T X$, $O(2^2 k)$ to multiply $\left(X^T X\right)^{-1}$ by X^T



(a) Sort 5G (b) InvertedIndex 5G

(c) Sort 10G (d) InvertedIndex 10G

(e) Sort 5G and 10G (f) InvertedIndex 5G and 10G

Fig. 4: Relationship between task duration and partition size

and finally $O(2k)$ to multiply $\left(X^T X\right)^{-1} X^T$ by Y. Therefore, the total computational complexity of the prediction model for a MapReduce job with N reduce tasks is $O(k \cdot N)$.

4.2 Reduce Task Performance Model

In this section, we design a reduce task performance model to estimate the execution time of reduce tasks. Currently, there are many techniques for predicting MapReduce job durations [12], [22], [23], [24]. These approaches, however, cannot estimate the durations of individual tasks. In our performance model we consider the execution time of a reduce task is correlated with two parameters: size of partition to process and resource allocation (e.g. CPU, disk I/O and bandwidth). As Hadoop YARN only allows users to specify the CPU and memory sizes of a container, in our implementation we focus on capturing the impact of CPU and memory allocations on task performance.

In order to identify the relationship between task running time, partition size and resource allocation, we run a set of experiments in our testbed cluster by varying resource allocation and input datasets. In the first set of experiments, we fix the CPU and memory allocations of each reduce task and focus on identifying the relationship between partition size and task running time. Figure 4a and 4b show the results of running the 5G Sort and InvertedIndex jobs, respectively. It is evident that there is a linear relationship between partition size and task running time. Hence, we use linear regression to determine this relationship with Equation 3 shown as follows:

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

where T_i and P_i are the running time and partition size of reduce task i, respectively. The regression results are also shown in Figures 4a and 4b as solid lines. Note that if the time complexities of the reduce functions in other MapReduce jobs grow nonlinearly with the sizes of processing data, the relationship can also be easily learned by updating the regression model.

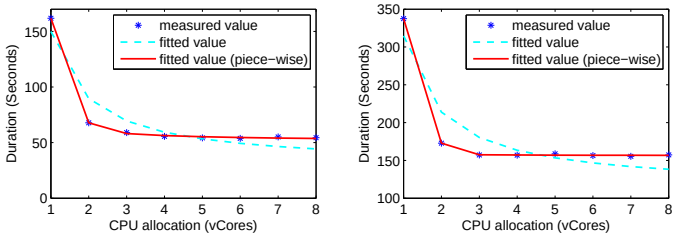
Furthermore, we change the input size of the jobs from 5GB to 10GB and check whether the characteristic of this relationship is workload independent. Again, the running time is linearly correlated with partition size, as shown in Figure 4c and 4d . However, we also find that the size of total intermediate data, denoted as D (the sum of all partitions), has an impact on task duration. Similar observation is also made in [22], where Zhang et al. show the duration of the shuffle phase can be approximated with a piece-wise linear function when the intermediate data per reduce task is larger than 3.2 GB in their Hadoop Cluster. This is consistent with the phenomenon we observed. Therefore, we update the regression function to Equation 4 and train the model by the samples from both 5GB and 10GB datasets together.

$$T_i = \alpha_2 + \beta_2 \cdot P_i + \zeta_2 \cdot D,\ i \in [1, N] \tag{4}$$

The regression results are shown in Figure 4e and 4f. It can be seen that this updated function serves as a good fit for the relationship between partition size and task running time, although there are two different datasets involved.

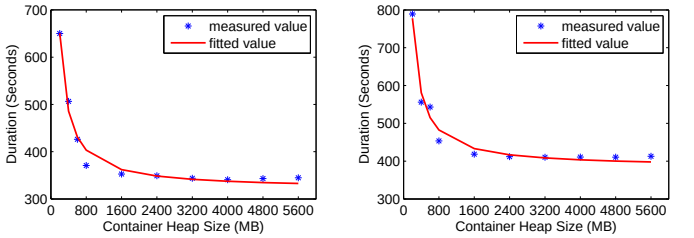
In the next set of experiments, we fix the input size and vary either the CPU or memory allocation of each reduce task. Figure 5 shows the typical results for 30G Sort and InvertedIndex jobs by varying CPU allocation from 1 to 8 vCores (memory allocation is fixed to 1 GB). We use a nonlinear regression method to model this relationship with Equation 5, and find that task running time is inversely proportional to CPU allocation. While this relationship fits well when the number of vCores is small, we also found this model is no longer accurate when a large amount of CPU resource is allocated to a task. In these cases, the resource bottleneck may switch from CPU to other resource dimensions like disk I/O, thus the benefit of increasing CPU allocation diminishes. Similar observation is also made in [24], where Jalaparti et al. show increasing network bandwidth beyond a threshold does not help since the job completion time is dominated by disk performance. This is consistent with the phenomenon we observed. Thus, we can expect that the duration of reduce tasks might be approximated with a different inversely proportional function when CPU allocation exceeds a threshold μ. This threshold could be related to job characteristics and cluster configuration. However, for a different job and Hadoop cluster, μ can be easily determined by comparing the change in task duration

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2532860, IEEE Transactions on Computers



(a) a reduce task in Sort (b) a reduce task in InvertedIndex

Fig. 5: Relationship between task duration and CPU allocation



(a) a reduce task in Sort (b) a reduce task in InvertedIndex

Fig. 6: Relationship between task duration and mem. allocation

while increasing CPU allocation.[1]

$$T_i = \alpha_3 + \frac{\beta_3}{A_i^{cpu}},\ i \in [1, N] \tag{5}$$

We then repeat the same set of experiments for memory. Different from the CPU allocation in YARN, which is determined by the number of virtual cores used by the task container, there are two configurations that control the memory allocation in YARN: physical RAM limit and JVM heap size limit for a task. The former setting is a logical allocation used by the Nodemanager to monitor the task memory usage. If the usage exceeds this limit, the Nodemanager will kill the task. The latter setting is maximum heap size of the JVM process that executes the task. It determines the maximum memory that can be used by this JVM. Hence, JVM heap size limit should be less than physical RAM limit. More importantly, JVM heap size indicates the amount of memory allocation that a task can use. Consequently, we vary the JVM heap size limit from 200 MB (the default value) to 5600 MB while keeping the CPU allocation to 1 vCore, and use a non-linear regression method to learn this relationship with Equation 6. We find that an inversely proportional function is also applicable in this case. Figure 6 shows the task running time as a function of memory allocation while running 30G Sort and InvertedIndex jobs. From this figure we can see an obvious improvement when the memory allocation increases at the beginning. That is because memory deficit will postpone the completion time of the task. For example, during the shuffle sub-phase in reduce stage, memory deficit will cause additional process of spilling data to disk, because of inadequate space to store all the data of a task in the RAM, thereby prolonging the task and adding a burden to disk I/O as well. However, with the allocation continually rising, the improvement becomes smaller. The reason is that, as memory allocation increases beyond a threshold, the resource bottleneck of a task shifts to other resources. After that point, the completion time of a task will not be reduced despite the increase in memory allocation. This observation is consistent with the CPU resource.

$$T_i = \alpha_4 + \frac{\beta_4}{A_i^{mem}},\ i \in [1, N] \tag{6}$$

Based on the above observations, we now derive our reduce task performance model. For each reduce task i among N reduce tasks, let T_i denote the execution time of reduce task i, P_i denote the size of partition for reduce task i, A_i^{cpu} denote the CPU allocation for reduce task i, and A_i^{mem} denote the memory allocation for reduce task i, the performance model can be stated as follows:

$$\begin{aligned} T_i &= (\alpha_5+\beta_5 P_i+\zeta_5 D)\cdot(\xi_5+\frac{\gamma_5}{A_i^{cpu}}+\frac{\eta_5}{A_i^{mem}}) \\ &= \alpha_5\xi_5+\frac{\alpha_5\gamma_5}{A_i^{cpu}}+\frac{\alpha_5\eta_5}{A_i^{mem}}+\beta_5\xi_5 P_i+\frac{\beta_5\gamma_5 P_i}{A_i^{cpu}}+\frac{\beta_5\eta_5 P_i}{A_i^{mem}} \\ &\quad +\zeta_5\xi_5 D+\frac{\zeta_5\gamma_5 D}{A_i^{cpu}}+\frac{\zeta_5\eta_5 D}{A_i^{mem}} \\ &= \lambda_1+\frac{\lambda_2}{A_i^{cpu}}+\frac{\lambda_3}{A_i^{mem}}+\lambda_4 P_i+\frac{\lambda_5 P_i}{A_i^{cpu}}+\frac{\lambda_6 P_i}{A_i^{mem}} \\ &\quad +\lambda_7 D+\frac{\lambda_8 D}{A_i^{cpu}}+\frac{\lambda_9 D}{A_i^{mem}} \end{aligned} \tag{7}$$

where λ_1, λ_2, λ_3, λ_4, λ_5 , λ_6, λ_7, $\lambda_8$and λ_9 are the coefficients to be solved using nonlinear regression. In practice, we may leverage historical records of job execution to provide input to the regression algorithm. This is reasonable in production environments as many jobs are executed routinely in today's production data centers. Specifically, the historical profiles are generated by varying CPU allocation $A_i^{cpu} = \{1\,vCore, 2\,vCores, \cdots, 8\,vCores\}$, memory allocation $A_i^{mem} = \{1\,GB, 2\,GB, \cdots, 4\,GB\}$, and input dataset $D_{set} = \{5\,GB, 30\,GB\}$ for different jobs. We then capture a tuple $(T_i,\ P_i,\ A_i^{cpu}\ A_i^{mem}, D)$ for each reduce task i of the job. Using the tuples for all reduce tasks as training data, we can easily learn the coefficient factors in the performance model for each job. In the end, we produce one performance model M_j (i.e. job profile) for each job j that can be used as an input for scheduling. Note that, if no job profile is available, DREAMS resorts to the default container allocation scheme (i.e. uniform container size for all the reduce tasks).

Finally, we would like to mention that while our performance model focuses on CPU and memory allocations, we believe our model can be extended to handle the case where other resources becomes the performance bottleneck by having additional terms in our performance model.

1. We use the following policy in this paper: we increase the CPU allocation from 1 to 8 vCores, and calculate the speedup of task running time between current and previous CPU allocations denoted as $Speedup_j$ ($j \in [1, 7]$). The first CPU allocation where $Speedup_j < 0.5 \cdot Speedup_{j-1}$ is considered as the threshold μ.

4.3 Resource Allocation Algorithm

Once the performance model has been trained and the partition size has been predicted, the scheduler is ready to find the optimal resource allocation to each reduce task so as to mitigate their run-time variation caused by partitioning skew. Here, our strategy is to equalize the running time of all reduce tasks. As mentioned in Section 4.2, task duration is a monotonically increasing function of partition size. Thus, we consider that the duration of the task with average partition size (P_{avg}) as a baseline denoted as T_{base}, which can be obtained according to Equation 7 with P_{avg} and the default CPU and memory allocations configured in YARN[2]. Then we increase the resources allocated to the reduce tasks with larger partition sizes to make them run no slower than T_{base}. We observed that there is also no need to allocate too much resources to large reduce tasks to make them run faster than T_{base}. Thus we wish to find the minimum CPU and memory allocations for enabling slower reduce tasks to meet the baseline T_{base}. It can be calculated using a variation of Equation 7 introduced in Section 4.2, where P_i, D and T_{base} are known.

$$\begin{aligned} T_{base} &= \lambda_1+\frac{\lambda_2}{A_i^{cpu}}+\frac{\lambda_3}{A_i^{mem}}+\lambda_4 P_i+\frac{\lambda_5 P_i}{A_i^{cpu}}+\frac{\lambda_6 P_i}{A_i^{mem}} \\ &\quad +\lambda_7 D+\frac{\lambda_8 D}{A_i^{cpu}}+\frac{\lambda_9 D}{A_i^{mem}} \end{aligned} \tag{8}$$

We can present Equation 8 in following form:

$$C_1+\frac{C_2}{A_i^{cpu}}+\frac{C_3}{A_i^{mem}}=0 \tag{9}$$

where $C_1=\lambda_1+\lambda_4 P_i+\lambda_7 D-T_{base}$, $C_2=\lambda_2+\lambda_5 P_i+\lambda_8 D$ and $C_3=\lambda_3+\lambda_6 P_i+\lambda_9 D$. Evidently, C_1, C_2 and C_3 are constants derived from known values. Since there are two variables (A_i^{cpu}, A_i^{mem}) needed to be solved using only one equation, more than one root can be obtained. In other words, there can be many possible CPU and memory combinations that will yield the same completion time, T_{base}. Hence, we formulate this resource allocation problem as a constrained optimization problem:

$$\begin{aligned} \min_{x,y} \quad & f(x_i, y_i) = x_i + \omega y_i \\ \text{s.t.} \quad & C_1+\frac{C_2}{x_i}+\frac{C_3}{y_i}=0 \\ & Cap_{cpu} > x_i > 1, \\ & Cap_{mem} > y_i > 1,\ i \in [1, N] \end{aligned} \tag{10}$$

where $x_i = A_i^{cpu}$, $y_i = A_i^{mem}$, and Cap_{cpu} and Cap_{mem} are the capacities of workers in terms of CPU and memory, respectively. We define the optimization function as the sum of CPU and memory resources, $x_i+\omega y_i$, where a factor ω is introduced for representing the weight of memory over CPU. We can configure a higher weight to the bottleneck resource that has lower availability. For instance, if CPU is lacking in the cluster but memory is not, CPU will become more "expensive" comparing to memory. In this case, increasing the weight of CPU can improve scheduling availability of tasks, thereby improving resource utilizations. ω depends on the capacity and the run-time resource availability of the cluster. How to tune ω is out of the scope of this work. In particular, we use $\omega = 1$ in this paper.

2. Here, since P_i can be predicted by the partition size prediction model, P_{avg} can be easily obtained. And the default CPU and memory allocations to a container in YARN are $1vCore$ and $1GB$, respectively.

Algorithm 1 Resource allocation algorithm

Input: δ - Threshold of stopping training the Partition Size Prediction Model;
M_j - Reduce Phase Performance Model of Job j;
μ_{cpu}, μ_{mem}- Maximum allowable allocation of CPU and memory.
Output: C - Set of resource allocations for each reduce task (A_i^{cpu}, A_i^{mem})

```
1: (S_i, F) ← handlePartitionReport().
2: if CompletedMap_percentage ≥ δ then
3:    Set < P_i > ← PredictPartition()
4:    D ← Σ_1^N P_i
5:    P_avg ← Avg(Set < P_i >)
6:    T_base ← PredictDuration(P_avg, D, A^cpu_default, A^mem_default, M_j)
7:    for each reduce task i ∈ [1, N] do
8:       (A_i^cpu, A_i^mem) ← FindOptimalAlloc(P_i, D, T_base, M_j).
9:       A_i^cpu = min(A_i^cpu, μ_cpu)
10:      A_i^mem = min(A_i^mem, μ_mem)
11:      C = C ∪ {(A_i^cpu, A_i^mem)}
12:   end for
13: end if
14: return C
```

Since this is a linear optimization problem, we use Lagrange multipliers to solve this problem. Accordingly, we get *the Lagrangian* $L(x_i, y_i, \varphi)$ as follows:

$$L(x_i, y_i) = x_i + \omega y_i + \varphi(C_1+\frac{C_2}{x_i}+\frac{C_3}{y_i}) \tag{11}$$

Then, we differentiate $L(x_i, y_i, \varphi)$ partially with respect to x_i, y_i and φ, and we get:

$$\begin{aligned} \frac{\partial L}{\partial x_i} &= 1-\varphi\frac{C_2}{x_i^2}=0 \\ \frac{\partial L}{\partial y_i} &= \omega-\varphi\frac{C_3}{y_i^2}=0 \\ \frac{\partial L}{\partial \varphi} &= C_1+\frac{C_2}{x_i}+\frac{C_3}{y_i}=0 \end{aligned} \tag{12}$$

Solving these equations simultaneously, we get:

$$x=-\frac{C_2 \pm\sqrt{\omega C_3 C_3}}{C1},\ y=-\frac{\omega C_3 \pm\sqrt{\omega C_3 C_3}}{\omega C1} \tag{13}$$

The detail of our resource allocation mechanism is shown in Algorithm 1. NodeManagers periodically send partition size reports to the ApplicationMaster along with heartbeat messages. ***As shown in Line 1, the ApplicationMaster handles each partition size report and collects the partition size statistics (S_i, F). Once the percentage of completed map tasks reaches the threshold δ, we start to predict the partition size and adjust the allocation for each reduce task as shown in Line 2-13. In terms of partition size prediction, we predict the partition size of each reduce task using the model presented in Section 4.1.*** With respect to the resource allocation, we compute the optimal combination of CPU and memory tuples (A_i^{cpu}, A_i^{mem}) using

TABLE 1: Benchmarks characteristics

Application	Domains	Dataset Type	Input Size Small (GB)	Skew-ness(%)	#Map, #Reduce	Input Size Large (GB)	Skew-ness(%)	#Map, #Reduce
WordCount	text retrieval	Wikipedia	5.759	14.75	92, 16	29.049	23.47	467, 64
BigramCount	text retrieval	Wikipedia	5.759	5.86	92, 16	29.049	12.62	467, 64
Pairs	text retrieval	Wikipedia	5.759	59.03	92, 16	29.049	55.04	467, 64
RelativeFreq	text retrieval	Wikipedia	5.759	46.37	92, 16	29.049	64.31	467, 64
InvertedIndex	web search	Wikipedia	5.759	16.94	92, 16	29.049	25.85	467, 64
AdjList	web search	GraphGenerator	5.012	0.88	90, 16	27.753	18.91	507, 64
KMeans	machine learning	Netflix	5.039	49.93	81, 6	26.412	49.81	428, 6
Classification	machine learning	Netflix	5.039	50.05	81, 16	26.412	49.99	428, 6
DataJoin	database	RandomTextWriter	5.556	36.46	81, 16	33.334	68.04	481, 64
SelfJoin	database	Synthetic	5.009	0.13	80, 16	27.99	0.22	448, 64
Sort	others	RandomWriter	5.086	31.52	85, 16	30.510	61.6	510, 64
Histo-movies	others	Netflix	5.039	234.98	81, 8	26.412	234.59	428, 8

Lagrange Multipliers. More specifically, we calculate the execution time T_{base} first, which represents the time it takes to complete the task with the average partition size P_{avg} and default resource allocation $(A^{cpu}_{default}, A^{mem}_{default})$[3], according to Equation 8. After that, we set T_{base} as a target for each reduce task, and calculate the resource tuples (A^{cpu}_i, A^{mem}_i) by solving Equation 13 and taking the floor of the positive root. Because nodes have finite resource capacities in terms of CPU and memory (e.g., the default settings for the maximum CPU and memory allocation to a container in YARN are 8 vCores and 8 GB, respectively), both A^{cpu}_i and A^{mem}_i should be less than the physical capacities, Cap_{cpu} and Cap_{mem}, respectively. Besides, from our experience, after a resource allocation to a task reaches a threshold, increasing allocation will not improve the execution time, rather it results in resource wastage as shown in Section 4.2. We consider A^{cpu}_i and A^{mem}_i should be less than the thresholds μ_{cpu} and μ_{mem}, respectively, which are considered as inputs to our algorithm.

5 Evaluation

We have implemented DREAMS on Hadoop YARN 2.4.0 as an additional feature. We deployed DREAMS on a real Hadoop cluster with 21 virtual machines (VMs) in the SAVI Testbed [25]. ***The SAVI Testbed is a virtual infrastructure managed by OpenStack [26] using Xen [27] virtualization technique.*** Each VM has four 2 GHz cores, 8 GB RAM and 80 GB hard disk. We use one VM as ResourceManager and NameNode, and the remaining 20 VMs as workers. Each worker is configured with 8 virtual cores and 7GB RAM (leaving 1GB for background processes). 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.[4]. ***We use CapacityScheduler to schedule containers in YARN. In the guest OS, we configure CGroups (Control Groups) and CFQ (Completely Fair Queuing) for scheduling CPU and disk I/O among processes, respectively.***

We evaluate our approach using a wide range of applications that include text retrieval, web search, machine learning, database domains, **etc.** ***These applications are listed below:***

1) Text Retrieval

- **WordCount (WC)**: WordCount computes the occurrence frequency of each word in a corpus. We use Wikipedia data as the input dataset.
- **BigramCount (BC)**: Bigrams are sequences of two consecutive words. BigramCount computes the occurrence frequency of bigrams in a corpus. We use the implementation in Cloud9 [30] and Wikipedia data as the input dataset.
- **Pairs (PS)**: "Pairs" is a design pattern introduced in [31]. Using this design pattern, PS computes the word co-occurrence matrix for a corpus. We use the implementation in Cloud9 and Wikipedia data as the input dataset.
- **RelativeFrequency (RF)**: Relative Frequencies is introduced in [31]. It measures the proportion of time word w_j appears in the context of word w_i. It is also denoted as $F(w_j|w_i)$. We use the implementation in Cloud9 and Wikipedia data as the input dataset.

2) Web Search

- **InvertedIndex (II)**: It takes a list of documents as input and generates a word-to-document index for these documents. We use Wikipedia data as the input dataset.
- **AdjacencyList (AL)**: It generates the adjacency list for a graph. The graph is represented by a set of edges, which is generated by a Graph Generator. We use the implementation and the input dataset provided by PUMA benchmarks [14].

3) Machine Learning

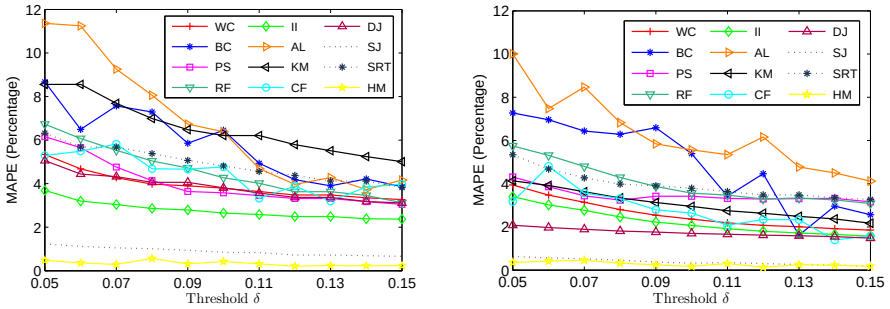
- **KMeans (KM)**: This application classifies movies based on their ratings using the Netflix movie rating data. We use the starting values of the cluster centroids provided by PUMA and run one iteration.
- **Classification**: It classifies the movies into one of k pre-determined clusters. Similar to KMeans, we use the starting values of the cluster centroids provided by PUMA, and use the Netflix movie rating data.

4) Database

- **DataJoin (DJ)**: It combines text files based on a designated key. The text dataset is generated by

3. The default CPU and memory allocations to a container are 1 vCore and 1 GB, respectively.

4. Using compression in Hadoop to optimize MapReduce performance is prevalent in industry and academia. [28], [29]



(a) Small dataset (b) Large dataset

Fig. 7: Prediction accuracy with different threshold δ

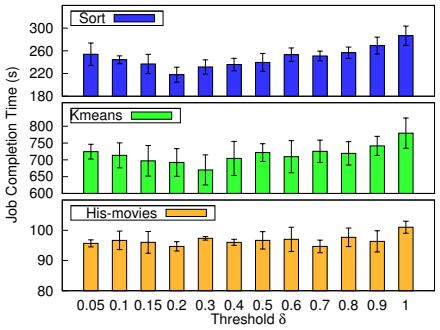


Fig. 8: Job completion time with different threshold δ

RandomTextWriter, and the first word of each line in the files serves as the join key. We have modified the original RandomTextWriter and used Zipf 0.5 distribution to skew the input data.

- **SelfJoin (SJ)**: This application is introduced in PUMA. It generates (k+1)-sized associations given the set of k-sized associations. We use the implementation of this application as well as the synthetic dataset in PUMA.

5) Others

- **Sort (SRT)**: This application sorts sequence files generated by Hadoop RandomWriter. Similar to [20], we have modified RandomWriter to produce non-uniformly distributed data.
- **Histogram-movies**: This application bins movies into 8 bins based on the average ratings of movies. We use the implementation of this application in PUMA.

Table 1 gives an overview of these benchmarks with their configurations used in our experiments. The skewness of the workload among reduce tasks is measured by the coefficient of variation (CV), $\frac{stdev}{mean}$, which is used as a fairness metric in literature [32]. The larger the ratio, the more skewness is expected in the distribution of workload among reduce tasks. In order to better demonstrate the skew mitigation, we do not use the combiner function in our benchmarks. We will present the results of running these jobs in the following sections.

5.1 Accuracy of Prediction of Partition Size

In this set of experiments, we want 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.

$$MAPE = \frac{1}{N}\sum_{i=1}^{N}\frac{\left|P_i^{pred} - P_i^{measrd}\right|}{P_i^{measrd}} \quad (14)$$

where N is the number of reduce tasks in a 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

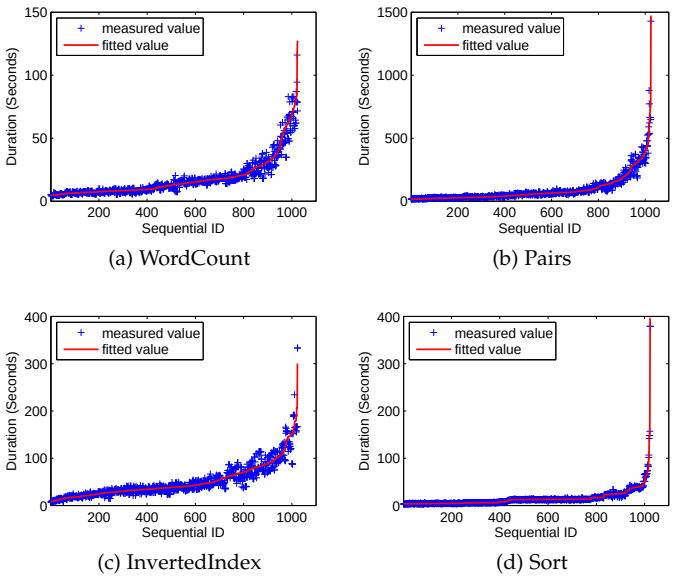
TABLE 2: Mean absolute percentage error of partition size prediction model on Small and Large datasets

Application	MAPE on Small dataset	MAPE on Large dataset
WordCount	5.34%	3.94%
BigramCount	8.67%	7.25%
Pairs	6.16 %	4.31 %
RelativeFrequency	6.73%	5.75%
InvertedIndex	3.69%	3.40%
AdjList	11.36%	10.01%
KMeans	8.56%	4.13%
Classification	5.29%	3.17%
DataJoin	5.06 %	2.08%
SelfJoin	1.23%	0.63%
Sort	6.32%	5.34%
Histogram-movies	0.47%	0.35%

MAPE for the benchmarks with threshold $\delta = 0.05$ on two different datasets. ***It can be seen that the error rates for most of the MapReduce applications are less than 5%. In particular, Adjlist reaches the highest error rate at 11.36%. Furthermore, Figure 7 illustrates the impact of different values of δ on prediction accuracy. It is clear that as δ increases, the prediction accuracy improves. That is because the number of training samples will augment along with the increase of δ. When $\delta = 0.15$, the prediction error achieves less than 6% for all testing applications.***

Generally speaking, increasing sample size can improve accuracy at the cost of increased overhead. In DREAMS, the larger the sample size used, the longer DREAMS has to wait for the completion of the map tasks for predicting the partition size[5]. However, we observed that as δ increases, the overhead in terms of job completion time does not necessarily become larger. Figure 8 shows the job completion times while using different values of δ. As shown in Figure 8, for reduce-intensive jobs such as Sort and Kmeans, there will be a sweet spot where the job completion time is lowest; for map-intensive jobs such as Histogram-movies, no much difference can be observed. The reason is that overlapping map and reduce phases can let the reduce task start to shuffle data earlier, but it will also waste resources while the map tasks' output rate is smaller than

5. The computational overhead is negligible, because the maximum number of samples is hundreds in our experiments.



(a) WordCount (b) Pairs

(c) InvertedIndex (d) Sort

Fig. 9: Fitting results for the reduce phase performance model

the bandwidth. Tang* et al. *[33] proposed a solution to find the best timing to start the reduce phase. This is out of the scope of this paper. In this paper, we use $\delta = 0.05$ ***in the following experiments.***

5.2 Accuracy of Reduce Task Performance Model

In order to formally evaluate the accuracy and workload independency of the performance model, we compute the prediction error using different datasets. That is, we train and test our model based on the samples from both the Small and Large datasets. ***Figure 9 shows the typical results in terms of the goodness of fit for the performance model. Similar results can be observed in other applications. To make the demonstration more clear, we sort the experiment results by the values in ascending order. The marks '+' represent the measured task durations and the solid line represents the fitted values using the performance model.***

We also perform two validations [34] to study the prediction accuracy of the model:

- **Resubstitution Method** - All the available data is used for training as well as testing. That is, we compute the predicted reduce task duration for each tuple $(P_i, Alloc_i^{cpu}, Alloc_i^{mem}, D)$ by using the performance model which is learned from the training dataset, then compute a prediction error;
- **K-fold Cross-validation** - The available data is divided into K disjoint subsets, $1 \leq K \leq m$. m is the total size of the available samples. And the prediction accuracy is evaluated by the average of the separate errors $\frac{1}{K}\sum_{i=1}^{K} Error_i$. For each of the K sub-validations, $(K-1)$ subsets are used for training and the remaining one for testing. Here, we choose $K = 10$.

TABLE 3: Mean absolute percentage error of the reduce phase performance model

Application	Resubstitution Method	K-fold Cross-validation
WordCount	13.13%	13.45%
BigramCount	10.26%	10.98%
Pairs	12.13 %	15.31 %
RelativeFrequency	13.03%	14.91%
InvertedIndex	12.97%	13.07%
AdjList	15.45%	18.02%
KMeans	12.52%	15.13%
Classification	4.61%	7.58%
DataJoin	7.84 %	14.09 %
SelfJoin	9.80 %	11.77 %
Sort	10.95%	11.46%
Histogram-movies	11.14%	14.46%

For both validations, we leverage the MAPE to evaluate the accuracy using following equation:

$$MAPE = \frac{1}{m}\sum_{l=1}^{m} \frac{\left|T_l^{pred} - T_l^{measrd}\right|}{T_l^{measrd}} \quad (15)$$

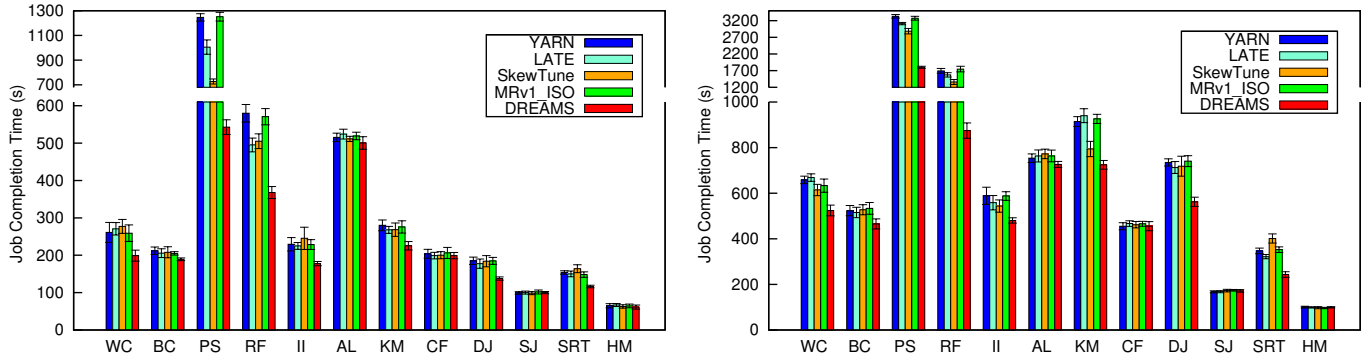
where m is the number of testing samples. Table 2 summarizes the MAPE of reduce task performance model for our testing workloads. ***With regard to the Resubstitution Method validation, the prediction error for all of the workloads is less than*** 15.45%***. In terms of the K-fold Cross-validation, the prediction error is slightly higher than the error in the Resubstitution validation. However, the error rate is still less than*** 18.02%***. For some applications such as Adjlist, the prediction error is relatively higher. But overall, the prediction error is less than*** 15% ***for most of the applications.*** Lastly, tuning the parameters of the performance model by continuously training the new coming data may improve the accuracy, which is considered as our future work.

5.3 Job Completion Time

In this section, we want to validate how well DREAMS can mitigate skew. We compare DREAMS against 1) Hadoop YARN 2.4.0; 2) Speculation-based straggler mitigation approach (LATE), which launches speculative tasks for the slower tasks; 3) repartition-based skew mitigation approach (SkewTune), which repartitions the unprocessed workload of the slower tasks at run-time and 4) Hadoop 0.21.0 with slot isolation (MRv1_ISO). To the best of our knowledge, in addition to SkewTune, many other state-of-the-art solutions such as LEEN [6], TopCluster [9], are implemented on top of MRv1, which is slot-based and there is no isolation between slots. In order to fairly compare DREAMS against SkewTune, we have implemented isolation between slots in Hadoop 0.21.0 and installed SkewTune on top of MRv1_ISO. We configure each worker with 6 map slots and 2 reduce slots while running SkewTune and MRv1_ISO. Note that tuning the number of reduce tasks of a MapReduce job can improve the job completion time [35]. To isolate this effect, we use the same number of reduce tasks in the corresponding experiments when comparing the job completion time.

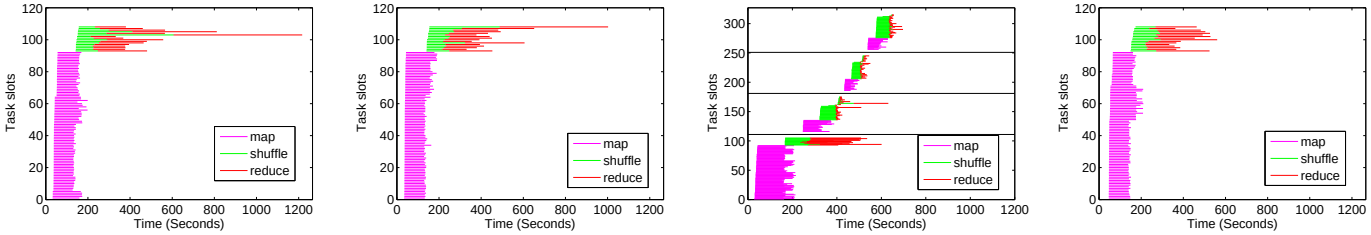
Figure 10 shows the comparison among YARN, LATE, SkewTune, MRv1 and DREAMS in regards to job completion time. We can see from the figure that DREAMS

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TC.2016.2532860, IEEE Transactions on Computers



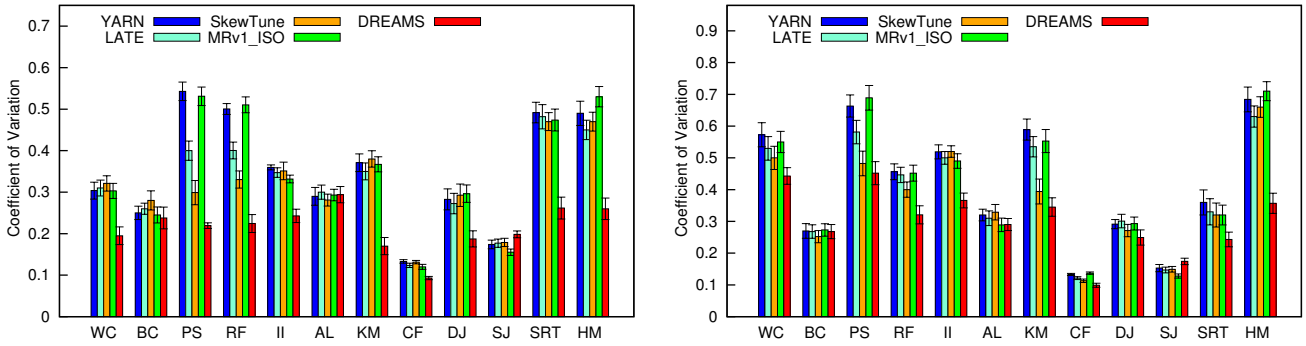
(a) Small dataset (b) Large dataset

Fig. 10: The comparison of job completion time



(a) YARN (b) LATE (c) SkewTune (d) DREAMS

Fig. 11: Execution Timeline for 5G Pairs



(a) Small dataset (b) Large dataset

Fig. 12: The comparison of makespan variance of reduce tasks

outperforms other skew mitigation strategies. In particular, DREAMS achieves $2.29\times$, $1.93\times$, $1.42\times$, $1.34\times$, $1.31\times$, $1.29\times$ ***and*** $1.26\times$ ***speedups over YARN for Paris, RelativeFreq, Sort, DataJoin, WordCount, InvtIndex and Kmeans, respectively. Compared to other mitigation strategies, DREAMS can achieve the highest improvements of*** $1.85\times$ ***and*** $1.65\times$ ***over LATE and SkewTune, respectively. We also observed that DREAMS cannot improve the job completion time for SelfJoin and Adjlist. This is because the skewness in these jobs is low, leaving little room for DREAMS and other mitigation strategies to improve. Since DREAMS only adjusts resource allocation for reduce tasks, for jobs such as Classification and HisMovies in which the reduce phase only lasts for a few seconds, no improvement in terms of the job completion time can be observed.***

In order to understand the reason behind the improvement of DREAMS, in Figure 11 we demonstrate the execution timeline while running 5G Pairs with YARN, LATE, SkewTune, and DREAMS, respectively. As shown in Figure 11a, several large reduce tasks take much longer than other reducers, which dominate the completion time of the job. In comparison, LATE executes replica tasks for these large reduce tasks using free resources, which can accelerate those large reduce tasks. However, since replica tasks process the same amount of work as original tasks, the improvement is not significant. SkewTune splits the unprocessed work

of stragglers at runtime, and launches new jobs (called migration jobs) to process these tasks. As we can see in Figure 11c, three additional jobs are launched to process those long lasting reduce tasks. Hence, overloaded tasks are processed using more cluster resources, which results in reducing their execution times. Note that the execution times of the stragglers are dominated by the completion times of the corresponding migration jobs. However, the overhead of repartitioning the running tasks is not small. As reported in [4], approximately 30s overhead is incurred for reduce skew mitigation, and SkewTune does not perform skew mitigation for the tasks with remaining time less than $2 \cdot w$, where w is on the order of 30s. As a results, for small jobs that complete in 100s or with small skewness, SkewTune cannot improve the job completion time. In contrast, DREAMS predicts the partition size of each reduce task at runtime and proactively allocates more resources to overloaded reducers. This reduces the durations of overloaded tasks, thereby accelerating the job completion with negligible overhead. As shown in Figure 11d, the running times of those large reducers are significantly improved.

We also compare the makespan variance of reduce tasks in DREAMS against other solutions. As we started earlier, DREAMS is designed to reduce the run-time difference among reduce tasks with different loads, thereby shortening the job completion time. ***Figure 12 shows the comparison results with respect to the coefficient of variation (CV) of reduce tasks' durations for our benchmarks. The graphs reflect that DREAMS can effectively reduce the makespan variance of reduce tasks. More specially, the highest reduction ratio can achieve*** $2.47\times$, $1.84\times$ ***and*** $2.23\times$ ***over YARN, LATE and SkewTune, respectively.*** Since the shuffle phase in reduce stage is overlapping the entire map stage, there is no need to count the makespan when the shuffle phase is waiting for the output of map tasks. Here, we compare the durations of reduce tasks starting from the completion of the last map task. ARIA [12] considers only the non-overlapping portions of shuffle into account. Chowdhury et al. [36] also define the beginning of the shuffle phase as when either the last map task finishes or the last reduce task starts.

6 Discussion

The concept of dynamic container size adjustment used in DREAMS is not restricted to MapReduce. It can be applied to other large-scale programming models such as Spark [37] and Storm [38] as well. Take Spark as an example, a Spark job consists of a number of tasks as a form of a DAG (Direct Acyclic Graph). These tasks are scheduled to a number of the Spark Executors and executed in a distributed manner. Each Spark executor runs in a container on top of the resource management platform (e.g. YARN and Mesos [39]). If some executors have more workload to process, dynamically adjusting the container size based on the resource requirements and workload characteristics may bring benefit as well.

Nevertheless, there are limitations in DREAMS's current design. First, DREAMS can only adjust CPU and memory for a container in our current implementation, and it relies on the TCP fairness and Completely Fair Queuing to fairly share the network bandwidth and disk I/O, respectively. Hence, DREAMS may not be able to give a precise estimation of the task execution time in a highly dynamic environment. However, our performance model works well in DREAMS. It roughly estimates the execution time of the reduce task based on the historical data and in turn helps DREAMS to determine how much resources should be allocated to the task. Through allocating more resources to the reduce tasks with more workload, the executions of these tasks can be accelerated, and therefore the job completion time can be improved. We would like to extend DREAMS to take account of network bandwidth and disk I/O in future work. One interesting idea is to integrate the management of containers' network and disk I/O resources to YARN using CGroups. Note that CGroups can support isolating network and disk I/O between processes currently. This deserves further research. Second, there may be some applications where DREAMS is not applicable, for example, the applications that contain computational skew [7] in their reduce functions. The computational skew refers to the case where the task running time depends on the content of the input rather than its size. For this kind of applications, DREAMS resorts to YARN in current design. One straightforward extension is to monitor the resource usage and progresses of tasks at run-time, and then adjust their allocation dynamically. In this way, skewed tasks could be accelerated in a more generic manner.

7 Related Work

The partitioning skew problem in MapReduce has been extensively investigated recently. The authors in [5] and [6] 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 for the completion of all the map tasks. Ramakrishnan *et al.* [7] and Yan *et al.* [19] propose to sample partition size before executing actual jobs to estimate the intermediate data distribution, and then partition the data to balance the load across all reducers. However, the additional sampling phase can be time-consuming. ***Similarly, Kolb et al. [40] propose two approaches, BlockSplit and PairRange, to handle data skew for entities resolution based on MapReduce. However, both of these two approaches have to run an additional MapReduce job to generate the block distribution matrix (BDM).*** Gufler *et al.* [9] and Chen *et al.* [10] propose to aggregate selected statistics of the key-value pairs (*e.g.* top k keys). Their solutions can reduce the overhead while estimating the reducer's workload, but these solutions still have to wait for the completion of all the map tasks. SkewTune [4] repartitions heavily skewed partitions at runtime to mitigate skew. However, it imposes an overhead while repartitioning data and concatenating final outputs. Compared to SkewTune, our solution dynamically allocates resources to reduce tasks and equalizes the reduce tasks' completion time, which is simpler and incurs no overhead.

There are also related works on culling stragglers in MapReduce. LATE [41] speculatively executes a replica task for the tasks at a slow progress rate. However, executing a redundant copy for a data-skew task, may result in wasting resource, since the duplicate tasks with data skew still have the same amount of data. Mantri [42] culls stragglers based

on their causes. With respect to data skew, Mantri schedules tasks in descending order of their input sizes to mitigate skew, which is complementary to DREAMS. Wrangler [43] predicts the status of worker nodes based on their runtime resource usage statistics, then selectively delays the execution of tasks if a node is predicted to create a straggler. However, Wrangler neglects that the straggling situation can also be incurred by the task itself; partitioning skew is one such example.

Resource-aware scheduling has received considerable attention in recent years. To address the limitation of slot-based resource allocation scheme in the first version of Hadoop, YARN [15] represents a major endeavor towards resource-aware scheduling in MapReduce. It offers the ability to specify the size of container. 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.* propose MROrchestrator [16], a MapReduce resource framework that can identify resource bottlenecks and resolve them by run-time resource allocation. However, MROrchestrator neglects the workload imbalance among tasks and cannot mitigate the partitioning skew. There are several other proposals that fall in other categories of resource scheduling policies such as [12], [18], [44], [45]. The main focus of those 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.

8 Conclusion

In this paper, we presented DREAMS, a framework for run-time partitioning skew mitigation. Unlike previous approaches that try to balance the reducers' workload by repartitioning the workload assigned to each reduce task, in DREAMS we cope with partitioning skew by adjusting run-time resource allocation to reduce tasks. Specifically, we first developed an on-line partition size prediction model which can estimate the partition size of each reduce task at run-time. We then presented a reduce task performance model that correlates run-time resource allocation and the size of the reduce task with task duration. In our experiments using a 21-node cluster running both real and synthetic workloads, ***we showed that both our partition size prediction model and task performance model achieve high accuracy in most cases (with highest prediction error at*** 11.36% ***and*** 18.02%***, respectively).*** We also demonstrated that DREAMS can effectively mitigate the negative impact of partitioning skew while incurring negligible overhead, thereby improving the job running time by up to a factor of 2.29 and 1.65 in comparison to the native Hadoop YARN and the state-of-the-art solution, respectively.

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] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[2] "Apache hadoop yarn," http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html.

[3] N. Zacheilas and V. Kalogeraki, "Real-time scheduling of skewed mapreduce jobs in heterogeneous environments," in *Proceedings of 11th International Conference on Autonomic Computing*. USENIX, 2014, pp. 189–200.

[4] 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*. ACM, 2012, pp. 25–36.

[5] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Handing data skew in mapreduce," in *Proceedings of the 1st International Conference on Cloud Computing and Services Science*, vol. 146, 2011, pp. 574–583.

[6] S. Ibrahim, H. Jin, L. Lu, B. He, G. Antoniu, and S. Wu, "Handling partitioning skew in mapreduce using leen," *Peer-to-Peer Networking and Applications*, vol. 6, no. 4, pp. 409–424, 2013.

[7] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing reducer skew in mapreduce workloads using progressive sampling," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 16.

[8] Y. Le, J. Liu, F. Ergun, and D. Wang, "Online load balancing for mapreduce with skewed data input," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 2004–2012.

[9] B. Gufler, N. Augsten, A. Reiser, and A. Kemper, "Load balancing in mapreduce based on scalable cardinality estimates," in *ICDE 2012*, April 2012, pp. 522–533.

[10] Q. Chen, J. Yao, and Z. Xiao, "Libra: Lightweight data skew mitigation in mapreduce," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 26, no. 9, pp. 2520–2533, 2015.

[11] L. Cheng, Q. Zhang, and R. Boutaba, "Mitigating the negative impact of preemption on heterogeneous mapreduce workloads," in *Proceedings of the 7th International Conference on Network and Services Management*. International Federation for Information Processing, 2011, pp. 189–197.

[12] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *Proceedings of the 8th ACM international conference on Autonomic computing*. ACM, 2011, pp. 235–244.

[13] Z. Liu, Q. Zhang, M. F. Zhani, R. Boutaba, Y. Liu, and Z. Gong, "Dreams: Dynamic resource allocation for mapreduce with data skew," in *IM 2015 - TechSessions*, Ottawa, Canada, may 2015.

[14] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "Puma: Purdue mapreduce benchmarks suite," 2012.

[15] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth *et al.*, "Apache hadoop yarn: Yet another resource negotiator," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 5.

[16] B. Sharma, R. Prabhakar, S. Lim, M. T. Kandemir, and C. R. Das, "Mrorchestrator: A fine-grained resource orchestration framework for mapreduce clusters," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 1–8.

[17] "Fair scheduler," http://hadoop.apache.org/docs/r2.4.0/hadoop-yarn/hadoop-yarn-site/FairScheduler.html.

[18] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types." in *NSDI*, vol. 11, 2011, pp. 24–24.

[19] W. Yan, Y. Xue, and B. Malin, "Scalable and robust key group size estimation for reducer load balancing in mapreduce," in *Big Data, 2013 IEEE International Conference on*. IEEE, 2013, pp. 156–162.

[20] M. Hammoud, M. S. Rehman, and M. F. Sakr, "Center-of-gravity reduce task scheduling to lower mapreduce network traffic," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 49–58.

[21] D. Borthakur, "The hadoop distributed file system: Architecture and design," *Hadoop Project Website*, vol. 11, p. 21, 2007.

[22] Z. Zhang, L. Cherkasova, and B. T. Loo, "Benchmarking approach for designing a mapreduce performance model," in *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. ACM, 2013, pp. 253–258.

[23] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics." in *CIDR*, vol. 11, 2011, pp. 261–272.

[24] V. Jalaparti, H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Bridging the tenant-provider gap in cloud services," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 10.

[25] "Smart applications on virtual infrastructure (savi)," http://www.savinetwork.ca/.

[26] "Open stack cloud operating system," http://www.openstack.org/.

[27] "Xen project," http://xenproject.org/.

[28] "Microsoft white papers: Compression in hadoop," http://technet.microsoft.com/en-us/library/dn247618.aspx.

[29] Y. Chen, A. Ganapathi, and R. H. Katz, "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*. ACM, 2010, pp. 23–28.

[30] J. Lin, "Cloud 9: A mapreduce library for hadoop," 2010.

[31] J. Lin and C. Dyer, "Data-intensive text processing with mapreduce," *Synthesis Lectures on Human Language Technologies*, vol. 3, no. 1, pp. 1–177, 2010.

[32] R. Jain, D.-M. Chiu, and W. R. Hawe, *A quantitative measure of fairness and discrimination for resource allocation in shared computer system*, 1984.

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

[34] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical pattern recognition: A review," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 1, pp. 4–37, 2000.

[35] Z. Zhang, L. Cherkasova, and B. T. Loo, "Autotune: Optimizing execution concurrency and resource usage in mapreduce workflows." in *ICAC*, 2013, pp. 175–181.

[36] M. Chowdhury, M. Zaharia, J. Ma, M. I. Jordan, and I. Stoica, "Managing data transfers in computer clusters with orchestra," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 98–109.

[37] "Apache spark," http://spark.apache.org/.

[38] "Apache storm," http://storm.apache.org/.

[39] "Apache mesos," http://mesos.apache.org/.

[40] L. Kolb, A. Thor, and E. Rahm, "Load Balancing for MapReduce-based Entity Resolution," in *International Conference on Data Engineering*, 2012, pp. 618–629.

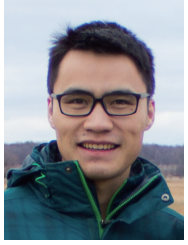
[41] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *OSDI*, vol. 8, no. 4, 2008, p. 7.

[42] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *OSDI*, vol. 10, no. 1, 2010, p. 24.

[43] N. J. Yadwadkar, G. Ananthanarayanan, and R. Katz, "Wrangler: Predictable and faster jobs using fewer resources," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[44] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *Network Operations and Management Symposium (NOMS), 2010 IEEE*. IEEE, 2010, pp. 373–380.

[45] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "Flex: A slot allocation scheduling optimizer for mapreduce workloads," in *Middleware 2010*. Springer, 2010, pp. 1–20.



Zhihong Liu received his 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 his 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.

Reaz Ahmed received his PhD in Computer Science from the University of Waterloo, Canada in 2007. His BSc. and MSc. degrees in Computer Science are from the Bangladesh University of Engineering and Technology (BUET), Dhaka, Bangladesh in 2000 and 2002, respectively. He is currently an Assistant Research Professor at the School of Computer Science in the University of Waterloo. His research interests include Network Virtualization, Network Function Virtualization, Software Defined Networking, Internet of things and Future Internet Architectures.

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 School of Computer with National University of Defense Technology. Her current research interests include network architecture, inter-domain routing, network virtualization and network security.

Zhenghu Gong received the B.E. degree in electronic engineering from Tsinghua University, Beijing, China, in 1970. He is currently a Professor in School of Computer with National University of Defense Technology, Changsha, China. His research interests include computer network and communication, network security and datacenter networking.