DREAMS: Dynamic Resource Allocation for MapReduce with Data Skew

Zhihong Liu*, Qi Zhang‡, Mohamed Faten Zhani‡, Raouf Boutaba‡ Yaping Liu† and Zhenghu Gong*
*College of Computer, National University of Defense Technology, Changsha, Hunan, China
Email:{zhliu,gzh}@nudt.edu.cn
†Science and Technology on Parallel and Distributed Processing Laboratory,
National University of Defense Technology, Changsha, Hunan, China
Email:ypliu@nudt.edu.cn
‡David R. Cheriton School of Computer Science, University of Waterloo, Waterloo, ON Canada
Email: {q8zhang,mfzhani,rboutaba}@uwaterloo.ca

***Abstract*—MapReduce has become a popular model for large-scale data processing in recent years. However, existing MapReduce schedulers still suffer from an issue known as partitioning skew, where the output of map tasks is unevenly distributed among reduce tasks. In this paper, we present DREAMS, a framework that provides run-time partitioning skew mitigation. Unlike previous approaches that try to balance the workload of reducers by repartitioning the intermediate data assigned to each reduce task, in DREAMS we cope with partitioning skew by adjusting task run-time resource allocation. We show that our approach allows DREAMS to eliminate the overhead of data repartitioning. Through experiments using both real and synthetic workloads running on a 11-node virtual virtualised Hadoop cluster, we show that DREAMS can effectively mitigate negative impact of partitioning skew, thereby improving job performance by up to** 20.3%**.**

I. Introduction

In recent years, the exponential growth of data in many application domains such as e-commerce, social networking and scientific computing, has generated tremendous needs for large-scale data processing. In this context, MapReduce [1] as a parallel computing framework has recently gained significant popularity. In MapReduce, a job consists of two types of tasks, namely *Map* and *Reduce*. Each map task takes a block of input data and runs a user-specified map function to generate intermediate key-value pairs. Subsequently, each reduce task collects intermediate key-value pairs and applies a user-specified reduce function to produce the final output. Due to its remarkable advantages in 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 still suffer from several important limitations. In particular, the most popular implementation of MapReduce, Apache Hadoop MapReduce [2], uses a hash function `Hash(HashCode(intermediate key) mod ReduceNumber)` 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 tasks, in reality this goal is rarely acheived [3]–[5]. For example, Zacheilas *et. al.* [3] have demonstrated the existence of skewness in Youtube social graph based on real workloads. The experiments showed that, the biggest size of partitions is larger than the smallest by more than a factor of five.

The skewed distribution of reduce workload can have severe consequences. First, data skewness may lead to a large difference in the runtime between the fastest and slowest tasks. As the completion time of a MapReduce job is determined by the finishing time of the slowest reduce task, data skewness can cause certain tasks to run much slower than others, thereby severely delaying job completion. Second, Hadoop MapReduce allocates fixed-size containers to reduce tasks. However, due to data skewness, different reduce tasks may have different run-time resource requirements. As a result, machines that are running tasks with heavy workload may experience resource contention, while machines with less data to process may experience resource idleness.

There are several approaches recently proposed to handle partitioning skew in MapReduce [4], [6]–[9]. Ibrahim et. al. proposed LEEN [6], a framework that balances reduce workload by assigning intermediate keys to reducers based on their record sizes. While this approach can mitigate the negative impact of data skew, its benefit is limited since the sizes of records corresponding to each key can still be unevenly distributed. Furthermore, it does not perform well when distribution of the records' sizes is severely skewed. Subsequently, Gufler et. al. [7] and Ramakrishnan et. al. [8] proposed techniques to split each key with large record size into sub-keys to allow for more even distribution of workload to reducers. However, most of these solutions have to wait until all the map tasks completed to gather the partition size information before reduce tasks can be started. The authors of [5], [9] demonstrate that by starting the shuffle phase after all map tasks are completed, the overall job completion time will be prolonged. While the progressive sampling [8] and adaptive partitioning [4] can eliminate this waiting time, the former approach requires an additional sampling phase to generate a partitioning plan before the job can be executed, whereas the latter approach incurs an additional run-time overhead (e.g. 30 seconds for certain jobs). In either case, the overhead due to repartitioning can be quite large for small jobs that takes from 10 to 100 seconds to complete. These small jobs are quite common in today's production clusters [10].

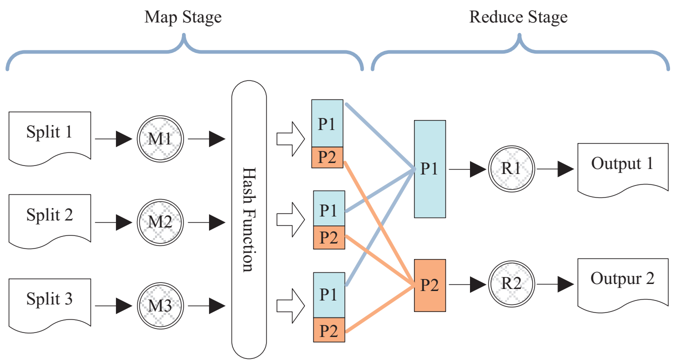


Fig. 1: MapReduce Programming Model

Motivated by the limitation of the existing solutions, in this paper, we take a completely different approach to address data skewness. Instead of subdividing keys into smaller sub-keys to balance the reduce workload, our approach adjusts run-time resource allocation of each reducer to match their corresponding data size. Since no repartitioning is involved, our approach completely eliminates the overhead due to repartitioning. To this end, we present DREAMS, a **D**ynamic **RE**source **A**llocation technique for **M**apReduce with data **S**kew. DREAMS leverages historical records to construct profiles for each job type. This is reasonable because many production jobs are executed repeatedly in today's production clusters [11]. At run-time, DREAMS can dynamically detect data skewness and assign more resources to reducers with large partitions to make them finish faster. In DREAMS, we first develop an online prediction model which can estimate the partition sizes of reduce tasks at runtime. We then establish a performance model that correlates run-time resource allocation with task completion time. Using this performance model, the scheduler can make scheduling decisions that allocate the right amount of resources to reduce tasks so as to equalize their running time. Through experiments using both real and synthetic workloads running on a 11-node virtualized Hadoop cluster, we show that DREAMS can effectively mitigate negative impact of partition skew, thereby improving job performance by up to 20.3%.

The rest of this paper is organized as follows. Section II provides the motivations of our work. We describe the system architecture of DREAMS in Section III. Section IV illustrates the design of DREAMS in detail. Section V provides the results of experimental evaluation. Finally, we summarize existing work related to DREAMS in Section VI, and draw our conclusion in Section VII.

II. Motivation

In this section we provide an overview of the partitioning skew problem and discuss the resource allocation issues in current MapReduce implementation therein motivating our study.

In state of the art MapReduce systems, each map task processes one split of input data, and generates a sequence of key-value pairs which are called intermediate data, on which hash partitioning function is performed. Since all map tasks use the same hash partitioning function, the key-value pairs with the same hash results are assigned to the same reduce task. In the reduce stage, each reduce task takes one partition (i.e. the intermediary key-value pairs received from all map tasks) as input and performs the reduce function on the partition to generate the final output. This is illustrated in Figure 1. Typically, the default hash function can provide load balancing if the key frequencies and the size of key-value pairs are uniformly distributed. This may fail with skewed data. For example in the InvertedIndex application, hash function partitions the intermediate data based on the words appeared in the file. Therefore, reduce tasks processing more popular words will be assigned a larger amount of data. As shown in Figure 1, partitions are unevenly distributed by the hash function. $P1$ is larger than $P2$, which causes workload imbalance between $R1$ and $R2$. [6] presents the causes of partitioning skew:

- *skewed key frequencies*: Some keys occur more frequently in the intermediate data, causing those reduce tasks that process these popular keys become overloaded.
- *skewed tuple sizes*: In applications where the sizes of values in the key-value pair vary significantly, uneven workload distribution may arise.
- *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 number of tuples per reduce task is the same, the execution times of reduce tasks may be different.

Due to many weaknesses and inadequacies experienced in the first version of Hadoop MapReduce (MRv1), the next generation of Hadoop compute platform, YARN [2], has been proposed. Nevertheless, in both Hadoop MRv1 and MRv2 (a. k. a. 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 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. On the other hand, YARN uses a container-based allocation scheme, where each task is scheduled in an isolated container with guaranteed CPU ad memory resources that can be specified in the request. But YARN still allocates containers of identical size to all reduce tasks that belong to the same job. In the presence of partitioning skew, this scheduling scheme can cause both variation in task running time and degradation in resource utilization. For instance, Kwon et. al. [4] demonstrated that in CloudBurst Application, there is a factor of five difference in runtime between the fastest and the slowest reduce tasks. Since the job completion time depends on the slowest task, the runtime variation of reduce tasks will prolong the job execution. At the same time, the reducers with large partitions run slowly because the resources allocated to them are limited by the container size, whereas reducers with light workload tend to under-utilize the resources allocated to the container. In both cases, the resulting resource allocation is inefficient.

Most of the existing approaches [4], [6]–[9] tackle the partitioning skew problem by making the workload assign-

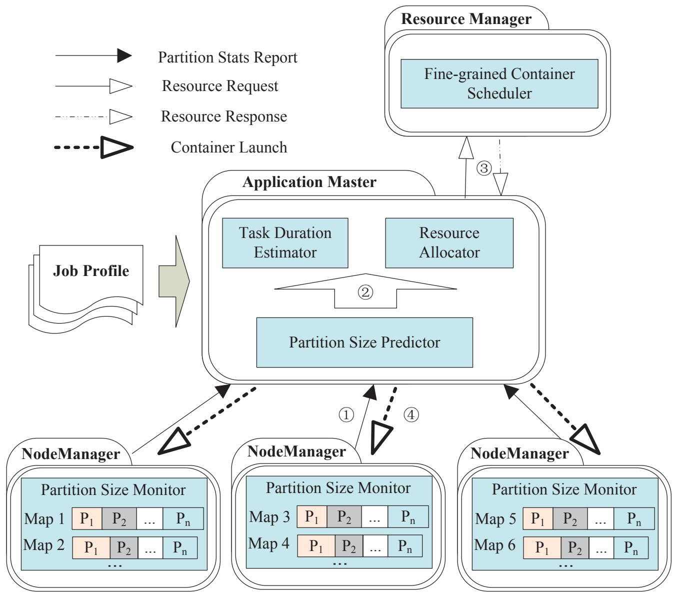


Fig. 2: Architecture of DREAMS

ment uniform 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, consisting in adjusting container size based on partitioning skew. This approach not only requires minimal modification to the existing Hadoop implementation, but at the same time can effectively mitigate the negative impact of data skew.

III. System Architecture

This section describes the design of our proposed resource allocation framework called DREAMS. The architecture of DREAMS is shown in Figure 2. Specifically, each *Partition Size Monitor* records the statistics of intermediate data that each map task generates at run-time and sends them to the ApplicationMaster though heartbeat messages. *Partition Size Predictor* collects the partition size reports from NodeManagers and predicts the partition sizes for this job at runtime. The *Task Duration Estimator* constructs statistical estimation model of reduce task performance as a function of its partition size and resource allocation. 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 task requests from ApplicationMasters according to scheduling policies such as Fair scheduling [12] and Dominant Resource Fairness (DRF) [13].

The workflow of resource allocation mechanism used by DREAMS 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 each reduce task. It then sends the statistics to the ApplicationMaster through the RPC protocol used to monitor the status of task 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 IV-A). The task Duration Estimator, which uses the job profiles (Section IV-B), predicts the task duration of each reduce task with specified amount of resources. Based on that, Resource Allocator determines the amount of resources for each reduce task according to our proposed resource allocation algorithm (Section IV-C) to equalize the execution time of all reduce tasks. (3) After that, the ResourceManager receives ApplicationMaster's resource requests through the heartbeat messages, and schedule free containers in the cluster to ApplicationMaster. (4) Once the ApplicationMaster obtains new containers from ResourceManager, it assigns the corresponding container to its pending task, and finally launches the task.

IV. DREAMS Design

There are two main challenges that need to be addressed in DREAMS. First, to identify partition skew, it is necessary to develop a run-time forecasting algorithm that predicts the partition size of each reducer. 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. In the following sections, we shall describe our technical solutions for each of the challenges.

A. Predicting Partition Size

As mentioned previously, the scheduler needs to know the partition size of each reduce task in order to compute the correct container size for that reduce task. Since current Hadoop schedulers allow reduce tasks to be launched soon after a fraction (e.g. 5%) of map tasks are finished[1], it is necessary to predict the partition size before the completion of all map tasks.

To predict the partition size of each reduce task i to be scheduled, at run-time the ApplicationMaster collects two metrics $\left(F^j,\, S_i^j\right)$, where F^j is the percentage of map tasks that have been processed, ($j \in [1, m]$ and m refers to the number of collected tuples $\left(F^j,\, S_i^j\right)$) and S_i^j is the size of the partition 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 completion. With these metrics, we use linear regression to determine the following equation for each reduce task $i \in [1, N]$:

$$a_1 + b_1 \cdot F^j = S_i^j \qquad j = 1,\, 2,\, \cdots\, m \tag{1}$$

We introduce an outer factor, δ, which is the threshold to control our prediction model to stop the process of learning, and finalize the prediction. In practice, δ can be the map completion percentage at which reduce tasks may be started to schedule (e.g. 5%). Every time a new map task has finished, a

[1]To improve job running time, existing Hadoop schedulers overlap the execution of map tasks and reduce tasks by allowing reduce tasks to be launched before the completion of all map tasks

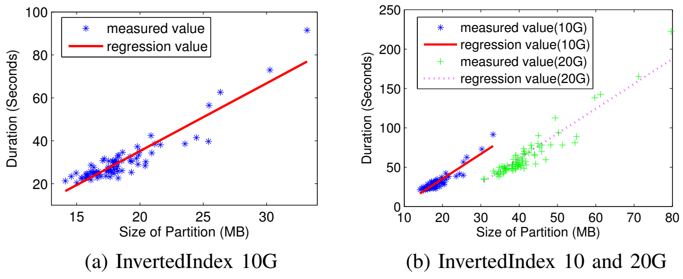


Fig. 3: Relationship between task duration and partition size

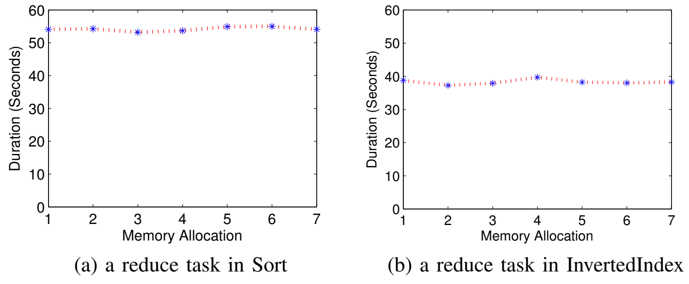


Fig. 5: Relationship between task duration and mem. allocation

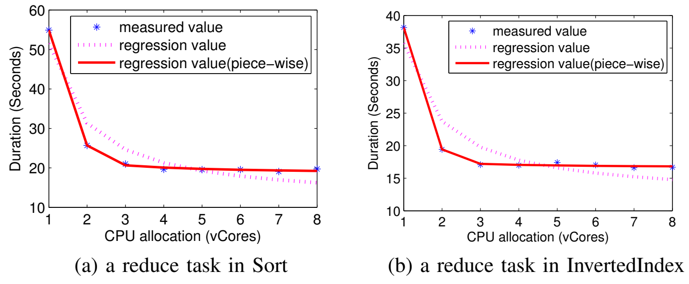


Fig. 4: Relationship between task duration and CPU allocation

new training data is created. When the fraction of map tasks reaches δ, we calculate the scaling factors (a_1, b_1) and predict the size of partition for each reduce task i of the whole data set, even though not all of the map tasks are completed.

We noticed that prediction schemes such as progressive sampling [8] can also be used by DREAMS for partition size prediction. However, the repartitioning mechanism used in [8] is based on a partitioning plan, and as result, it requires progressive sampling to be executed each time before the job starts. In our case, since we do not need to modify the implementation of partitioning, our partition size prediction can be done entirely online. Thus, we found our current prediction scheme is simple yet sufficient to produce high quality prediction results.

B. Reduce Phase Performance Model

In this section, we design a task performance model that correlates the completion time of individual reduce tasks with their partition size and resource allocation. As Hadoop YARN only allows the CPU and memory size of container to be specified, in our implementation we focus on capturing the impact of CPU and memory allocation on task performance.

In order to identify the relationship between task running time, partition size and resource allocation, we run a set of benchmarks in our Testbed cluster by varying resource allocation. More specifically, each benchmark is generated by varying CPU allocation $Alloc_i^{cpu} = \{1\,vCore, 2\,vCores, \cdots, 8\,vCores\}$, memory allocation $Alloc_i^{mem} = \{1\,GB, 2\,GB, \cdots, 8\,GB\}$, and input dataset $D_{set} = \{10\,GB, 20\,GB, 30\,GB, 50\,GB\}$ for different jobs. We run each benchmark 10 times, and collect the average result over the runs for each benchmark.

In the first set of experiments, we fix the CPU and memory allocation of each reduce task and focus on identifying the relationship between partition size and task running time. To illustrate, Figure 3a shows the result of running the InvertedIndex job using 10GB input. It is evident that there is a linear relationship between partition size and running time. Furthermore, Figure 3b shows the result when the input size of the job is changed from 10GB to 20GB. Again, the running time is linearly correlated with partition size. However, at the same time, we also found that the size of total intermediate data denoted as D (the sum of all partitions) has an impact on task duration while varying the input dataset. Similar observation is also made in [14], 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 that 3.2 GB in their Hadoop Cluster. This is consistent with the phenomenon we observed.

In the next set of experiments, we fix the input size and vary either the CPU or memory allocation of each reduce task. Figure 4 shows the typical results for Sort and InvertedIndex job by varying the CPU allocation (memory allocation is fixed to 1 GB). We found that task running time is inversely proportional to CPU allocation. In particular, the task running time is approximately halved when the CPU allocation is increased from 1 vCore to 2 vCores. While this relationship is accurate 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, in which case the benefits of increasing CPU allocation would decrease. 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 φ. This threshold could be related to Job characteristics and cluster configuration. However, for a different Job and Hadoop cluster, φ can be easily determined by comparing the change in task duration while increasing CPU allocation.[2]

We then repeat the same experiment for memory;we vary the memory allocation from 1 to 7 GB while the CPU is fixed to 1 vCore. We found the same relationship does not apply to memory. Figure 5 shows the task running time as

[2]We use the following policy in this paper: we increase the CPU allocation from 1 vCore to 8 vCores, and caluate the speedup of task running time between current and previous CPU allcoations 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 φ.

a function of memory allocation. We found that even though memory allocation is increased, no improvement can be found. We believe the reason is that memory is not the bottleneck resource for this task. In this case, the memory allocation will not affect task duration as long as it is sufficient for this task.

Based on the above observations, we now derive our 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, D denote the size of the intermediate data for the job, and $Alloc_i^{cpu}$ denote as the CPU allocation for reduce task i, the performance model can be stated as:

$$\begin{aligned} &When\, Alloc_i^{cpu} <= \varphi, \\ &T_i = \alpha + \beta P_i + \gamma D + \frac{\zeta}{Alloc_i^{cpu}} + \frac{\eta P_i}{Alloc_i^{cpu}} + \frac{\xi D}{Alloc_i^{cpu}} \\ &When\, Alloc_i^{cpu} > \varphi, \\ &T_i = \alpha' + \beta' P_i + \gamma' D + \frac{\zeta'}{Alloc_i^{cpu}} + \frac{\eta' P_i}{Alloc_i^{cpu}} + \frac{\xi' D}{Alloc_i^{cpu}} \end{aligned} \tag{2}$$

where $\alpha, \beta, \gamma, \zeta, \eta, \alpha', \beta', \gamma', \zeta'$ and η' are the coefficient factors to be solved using nonlinear regression [15]. 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, we capture a triple $(T_i, P_i, Alloc_i^{cpu})$ for each reduce task i of the job. Using the triples 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 for each job j that can be used as input for scheduling.

Finally, we would like to mention that while our performance model focuses on CPU allocation, we believe our model can be extended to handle the case where other resources becomes the performance bottleneck by having additional terms (e.g. similar to the second and third term in equation 2) in our performance model.

C. Scheduling Algorithm

Once the performance model has been trained and the partition size has been predicted, the scheduler can now decide how much resource to be allocated to each task. In order to mitigate the impact of data skew, we adopt a simple strategy which is to make all reduce tasks have similar running time. Algorithm 1 describes our resource allocation policy. After reaching the threshold δ, the partition size of each reduce task can be predicted with the prediction model. As to memory allocation, it does not affect task duration as long as it is sufficient for this task, which is discussed in Section IV-B. We adjust the memory allocation to $\lceil \frac{P_i}{Unit^{mem}} \rceil \cdot Unit^{mem}$, where $Unit^{mem}$ is the minimum allocation of memory. With respect to CPU allocation, we obtain the amount of resources according to performance model M_j, as described from line 5 to line 12. First, we calculate the execution time T_{mid} , which represents the time it takes to complete the task with the median partition size P_{mid}, by performance model M_j. After that, we set T_{mid} as target for each reduce task, and calculate the amount of resources $Alloc_i^{cpu}$ that each reduce

Algorithm 1 Resource allocation algorithm

```
Input: $\delta$ - Threshold of stopping training the Partition Size Prediction
    Model; $M_j$ - Reduce Phase Performance Model of Job $j$; $\varphi$ -
    Maximum allocation of CPU;
Output: $C$ - Set of resource allocations for each reduce task
    $\langle Alloc_i^{cpu}, Alloc_i^{mem} \rangle$
 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 $Set < P_i >$ while $F = 100\%$
 5: Calculate the median value $P_{mid}$ in $Set < P_i >$
 6: Calculate $T_{mid}$, when $P = P_{mid}$, $Alloc^{cpu} = 1vcore$ using $M_j$
 7: for each reduce task $i \in [1, N]$ do
 8:     $Alloc_i^{mem} = \lceil \frac{P_i}{Unit^{mem}} \rceil \cdot Unit^{mem}$;
 9:     Solve the Equation 2 for $Alloc_i^{cpu}$
10:     if $Alloc_i^{cpu} \geq \varphi$ then
11:         $Alloc_i^{cpu} = \varphi$
12:     end if
13:     $C = C \cup \{\langle Alloc_i^{cpu}, Alloc_i^{mem} \rangle\}$
14: end for
15: return $C$
```

task needs. Because nodes have finite resource capacities, $Alloc_i^{cpu}$ should be less than the capacities. Besides, from our experience, after CPU allocation to a task reaches a threshold, increasing allocation will not improve the execution time, but instead results in wasting CPU resource as shown in Section IV-B. We consider $Alloc_i^{cpu}$ should be less than threshold φ, which is also an input to our algorithm.

V. Evaluation

We perform our experiments on 11 virtual machines (VMs) in the SAVI Testbed [16], 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 Resource Manager and Name Node, and remaining 10 VMs as workers. Each worker is configured with 8 virtual cores and 7GB RAM (leaving 1GB 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 128MB, and the replication level is set to 3.

We chose two jobs to evaluate DREAMS: (1) *Sort*, which is included in a MapReduce benchmark in Hadoop distribution. It takes sequence files which are generated by RandomWriter as input, and outputs the sorted data, and (2) *InvertedIndex*, which comes from PUMA benchmarks [17]. It takes a list of documents as input and generates an inverted index for these documents. We use Wikipedia data [17] for this application.

A. 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 with different thresholds δ, and compute the average relative error (ARE) of all partitions in each scenario. The ARE is defined as follows.

$$ARE = \frac{1}{N} \sum_{i=1}^{N} \frac{\left| P_i^{pred} - P_i^{measrd} \right|}{P_i^{measrd}} \tag{3}$$

TABLE I: Average relative error of partition size prediction

Application	Input Data Type	Input Data Size(GB)	ARE $\delta = 0.05$	ARE $\delta = 0.06$	ARE $\delta = 0.07$	ARE $\delta = 0.08$	ARE $\delta = 0.09$	ARE $\delta = 0.10$
Sort	Synthetic	10	2.28%	2.09%	1.94%	1.81%	1.71%	1.71%
Sort	Synthetic	20	1.60%	1.43%	1.32%	1.26%	1.17%	1.13%
Sort	Synthetic	50	1.1%	1.01%	0.94%	0.90%	0.84%	0.78%
InvertedIndex	Wikipedia	9.01	8.2%	7.63%	7.05%	7.05%	6.43%	5.87%
InvertedIndex	Wikipedia	21.02	5.62%	5.25%	5.08%	4.79%	4.53%	4.38%
InvertedIndex	Wikipedia	49.04	4.73%	4.43%	4.21%	4.07%	3.90%	3.70%

where N is the number of reduce tasks in this job, P_i^{pred} and P_i^{measrd} are the predicted value and measured value of partition size of reduce task i respectively. Table I summarizes the average relative errors in each scenario. We run 10 experiments for each scenario and adopt the average. It can be seen that the ARE is less than 8.2% in all cases. Furthermore, with threshold δ increases, the prediction accuracy is improved.

B. Accuracy of reduce phase performance model

In order to formally evaluate the accuracy and workload independency of the generated performance model, we compute the prediction error for Sort and InvertedIndex with different input workloads. We perform two validations as follow:

- **Test-on-training** - evaluate the accuracy of preformance model based on the training dataset. That is, we compute the predicted reduce task duration for each tuple $(P_i, Alloc_i^{cpu})$[3] by using the performance model which is learned from this training dataset, then compute a prediction error;
- **Test-on-unknown** - evaluate the accuracy of performance model using unknown dataset. That is, we compute the predicted reduce task duration for each tuple $(P_i, Alloc_i^{cpu})$ by using the performance model which is learned from 10 G workload (This derived model is considered as a profile), then compute a prediction error.

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

$$ARE = \frac{1}{k}\sum_{l=1}^{k} \frac{\left|T_l^{pred} - T_l^{measrd}\right|}{T_l^{measrd}} \qquad (4)$$

where k is the number of tuples $(P_i, Alloc_i^{cpu})$ for a input dataset. Table II summarizes the average relative error of reduce task performance model for Sort and InvertedIndex. More specifically, with regard to Test-on-training validation, the prediction error for Sort and InvertedIndex with all of the workloads is less than 15%. For the Test-on-unknown group, the prediction error is slightly higher than the corresponding value in the Test-on-training, still less than 20%. These results confirm the accuracy of our performance model.

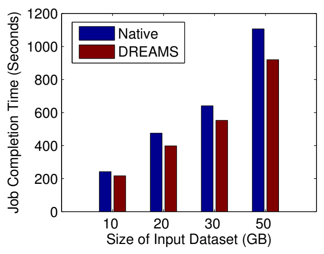
C. Performance Evaluation

We have implemented DREAMS on Hadoop YARN 2.4.0 as an additional feature. Implementing this approach requires

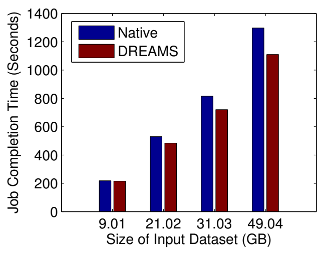
[3]For example, there are N reduce tasks of a job, for each reduce task i, there are one value of P_i and 8 values of $Alloc_i^{cpu} \in \{1, 2, \cdots, 8\}$. Therefore, there are $8N$ tuples for this workload.

TABLE II: Average relative errors of reduce task performance model

Application	Input Data Type	Input Data Size(GB)	Test-on-training	Test-on-unknown
Sort	Synthetic	10	5.44%	9.36%
Sort	Synthetic	20	7.91%	10.62%
Sort	Synthetic	30	12.28%	16.38%
Sort	Synthetic	50	11.09%	19.57%
InvertedIndex	Wikipedia	9.01	11.67%	13.97%
InvertedIndex	Wikipedia	21.02	12.89%	13.31%
InvertedIndex	Wikipedia	31.03	14.67%	16.44%
InvertedIndex	Wikipedia	49.04	14.56%	17.06%

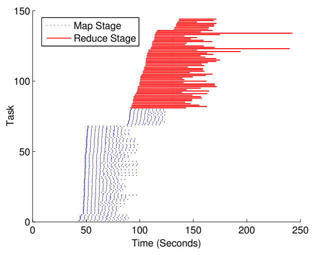


(a) Sort

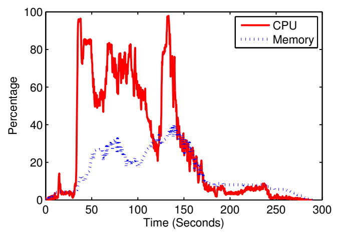


(b) InvertedIndex

Fig. 6: Job completion time of individual jobs

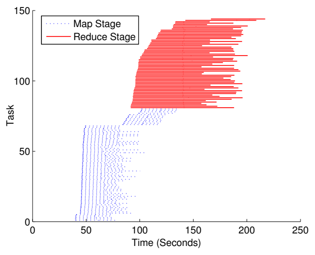


(a) Task Execution Timeline

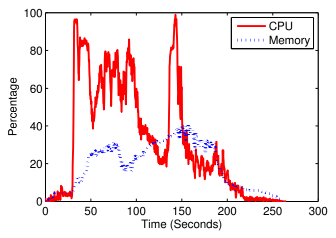


(b) CPU and Mem. Util.

Fig. 7: Sorting 10GB with Native Hadoop



(a) Task Execution Timeline



(b) CPU and Mem. Util.

Fig. 8: Sorting 10GB with DREAMS

TABLE III: Workloads characteristics

Application	Input Data Type	Input Data Size(GB)	CV of Partition Sizes	#Map and Reduce tasks
Sort	Synthetic	10	46.24%	80,64
Sort	Synthetic	20	46.24%	160,64
Sort	Synthetic	30	46.24%	240,64
Sort	Synthetic	50	46.24%	400,64
InvertedIndex	Wikipedia	9.01	17.44%	73,80
InvertedIndex	Wikipedia	21.02	19.12%	169,80
InvertedIndex	Wikipedia	31.03	22.13%	252,80
InvertedIndex	Wikipedia	49.04	24.95%	396,80

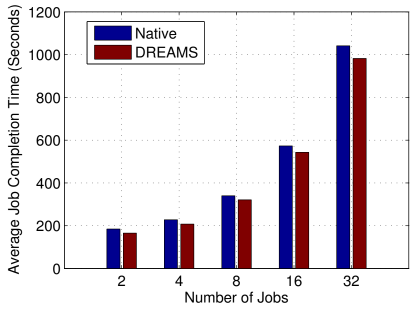
minimal change to the existing Hadoop architecture. In this section, we compare the performance of DREAMS against native Hadoop YARN 2.4.0 (called Native in this paper). The *slowstart* threshold is set to 10%, and the *CgroupsLCEResourcesHandler* is enabled. We first evaluated DREAMS using individual Job (either Sort or InvertedIndex) with several input data sizes from 10 GB to 50 GB. Table III gives an overview of these workloads. Note that tuning the number of reduce tasks for each workload can improve job completion time [18]. To isolate this effect, we fix the number of reduce tasks for each job. The CV (coefficient of variation) of partition sizes represents the skewness of the reduce input distribution. We can see from the table that the CV values of all the workloads are less than 50%[4]. The experiment results are shown in Figure 6. We can see from the figure that DREAMS outperforms Native for all cases. In particular, DREAMS improves job completion time by 20.3% when sorting 50 GB data. To understand the reason behind the performance gain, we plotted the timeline and cluster CPU and memory usage of executing 10G Sort for Native and DREAMS in Figure 7 and Figure 8. We found that DREAMS equalizes the durations among reduce tasks, and achieves higher CPU and Memory utilization than Native in reduce stage. More specifically, the utilization between DREAMS and Native during map stage is similar; after map stage completes (around 150 seconds mark), both CPU and memory utilization of DREAMS become higher than Native. Furthermore, we have found that DREAMS generally achieves higher reduction in job completion time for Sort rather than InvertedIndex. That is because DREAMS only improves the resource allocation in reduce stage, but leaving map stage unchanged. And Sort is reduce intensive, where reduce stage takes longer time than map stage. As a result, DREAMS is able to provide higher gain for the job running time of reduce-intensive jobs.

We now present our evaluation results using multiple jobs in parallel. According to the cumulative distribution function of job running times from a production workload trace at Facebook [19], the job completion times follow a long-tail distribution. More specifically, most of the jobs (more than 50%) are less than 100 seconds long, and the distribution of inter-arrival times for this workload trace is roughly exponential with a mean of 14 seconds. Therefore, in this evaluation, we vary the number of jobs of 5G Sort and 5G InvertedIndex from $\times 1$ to $\times 16$ to create batch workloads, and submit the jobs with an inter-arrival time following exponential distribution with a mean of 14 seconds. We run each of the batch workloads 5 times using Native and DREAMS. The results of average job completion time are shown in Figure 9a. It can be seen that DREAMS outperforms Native Hadoop in all scenarios. Admittedly, the gain of DREAMS in experiments with multiple jobs is less than the gain in single job experiments. It is because reduce tasks of small jobs only last dozens of seconds, which means the difference between the longest and shortest task is only dozens of seconds. When the number of jobs is increasing, many short tasks are scheduled one after the other. As a result, there is a chance that these short tasks can fit into resource vacancy that skewed tasks generate. Therefore, in some cases DREAMS can obtain only dozens of seconds gain for these small jobs (Note that dozens of seconds constitute a big gain in single job scenario). In the future, we intend to evaluate DREAMS using multiple large jobs. Figure 9b and 9c show the resource utilization of the cluster during the execution of each batch for Native and DREAMS respectively. It can be seen from the diagrams that DREAMS achieves slightly higher CPU utilization than the Native Hadoop, and the memory utilizations of both methods are similar. That is because the biggest partition size of all the reduce tasks in this workload is less than the minimal memory allocation, DREAMS does not adjust the memory. But with respect to CPU allocation, DREAMS makes an adjustment for different reduce tasks, thereby achieving higher CPU utilization.

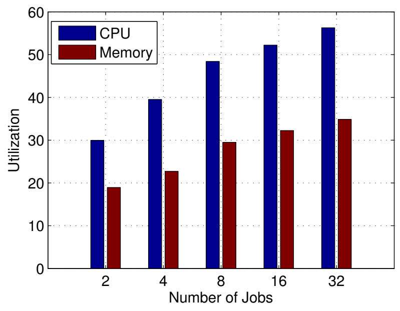
VI. Related Work

The data skew problem in MapReduce has been extensively investigated recently. Kwon et.al. [20] present five types of skews in MapReduce applications which are caused by the characteristics of the algorithm and dataset, and propose best practices to mitigate skew. On mitigating the impact of skewed data, several approaches have been proposed. The authors, in [7] and [6], define a cost model for scheduling 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 [5], 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 [9] applies a Greedy-Balance approach of assigning unassigned 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. Even though the results in this paper show a reduction of maximum load compared to default solution, shuffle finishing time is worse than the default solution. Also this paper provides no evaluation about whether the job completion time can be shortened. Unlike those later shuffling approaches, Ramakrishnan et.al. [8] 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 [4] repartitions heavily skewed partitions to achieve this goal. However, it imposes an overhead while repartitioning data and concatenating original output. Compared to SkewTune, our solution dynamically allocates the right amount of resources to tasks to equalize the tasks' completion time, which is simpler and incurs no overhead. Finally, Zacheilas et al. propose DynamicShare [3], which aims at scheduling MapReduce jobs in heterogeneous systems to meet their real-time response requirements, and achieving an

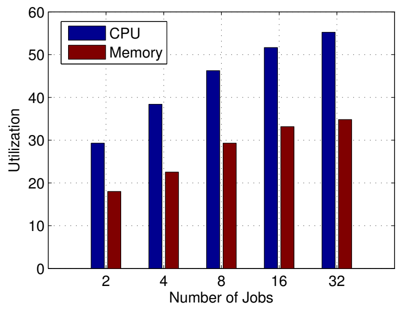
[4]The CV of each workload for Sort is the same, because these workloads are generated by the same RandomWriter.



(a) Job Completion Time



(b) Resource Util. of DREAMS



(c) Resource Util. of Native

Fig. 9: Multiple Jobs Benchmark

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 partitions' assignment procedure. Besides, DynamicShare cannot start partitions assignment until all map tasks have completed.

Resource-aware scheduling has received considerable attention in recent years. The original Hadoop MapReduce implements a slot-based resource allocation scheme, which does not take run-time task resource consumption into consideration. To address this limitation, Hadoop YARN [2] 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. propose MROrchestrator [21], a MapReduce resource framework that can identify the resource deficit based on resource profiling, and dynamically adjusts the resource allocation. Compared with our solution, MROrchestrator cannot identify stragglers of workload imbalance before tasks launch, and it cannot judiciously place tasks that need more resource on the machines with more free resources. In other words, if all CPU-intensive tasks are launched in a machine, no matter how 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 [11], [13], [22], [23]. 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.

VII. Conclusion

MapReduce has become a predominant model for large-scale data processing in recent years. However, existing MapReduce schedulers still use a simple hash function to assign map outputs to reduce tasks. This simple data assignment scheme may result in a phenomenon known as partitioning skew, where the output of map tasks is unevenly distributed among reduce tasks. While many approaches have been proposed to address this issue, existing solutions often incur an additional overhead for run-time partition size prediction and data repartitioning. Motivated by this limitation, in this paper we present DREAMS, a framework for run-time partitioning skew mitigation. Unlike previous approaches that try to balance the reduce workload by repartitioning the workload assigned to each reduce task, in DREAMS we cope with partitioning skew by adjusting task run-time resource allocation. To do so, we first develop an on-line partition size prediction model which can estimate the partition sizes of reduce tasks at run-time. Our experiments results show that the average relative error is less than 8.2% in all cases. Second, we design a reduce task performance model that correlates task duration with run-time resource allocation and input size of reduce tasks. The validation results show that the worse prediction error is 19.57%. Third, we demonstrate the benefit of leveraging resource-awareness for run-time skew mitigation. Through experiments using real and synthetic workloads, we show that DREAMS can effectively mitigate the negative impact of partitioning skew while incurring negligible overhead, thereby improving job running time by up to 20.3%.

Acknowledgement

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] 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.

[3] A. Computing, N. Zacheilas, and V. Kalogeraki, "Real-time scheduling of skewed mapreduce jobs in heterogeneous environments."

[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] 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.

[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] 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.

[8] 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.

[9] Y. Le, J. Liu, F. Ergun, and D. Wang, "Online load balancing for mapreduce with skewed data input."

[10] 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.

[11] 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.

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

[13] 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.

[14] 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.

[15] D. M. Bates and D. G. Watts, *Nonlinear regression: iterative estimation and linear approximations*. Wiley Online Library, 1988.

[16] J.-M. Kang, H. Bannazadeh, and A. Leon-Garcia, "Savi testbed: Control and management of converged virtual ict resources," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*. IEEE, 2013, pp. 664–667.

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

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

[19] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 265–278.

[20] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "A study of skew in mapreduce applications," *Open Cirrus Summit*, 2011.

[21] 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.

[22] 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.

[23] 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.