Computer Modeling in Engineering & Sciences

Tech Science Press

DOI: 10.32604/cmes.2021.013244

ARTICLE

Run-Time Dynamic Resource Adjustment for Mitigating Skew in MapReduce

Zhihong Liu[1], Shuo Zhang[2,*], Yaping Liu[2], Xiangke Wang[1] and Dong Yin[1]

[1]College of Intelligence and Technology, National University of Defense Technology, Changsha, 410073, China

[2]Cyberspace Institute of Advanced Technology, Guangzhou University, Guangzhou, 510006, China

[*]Corresponding Author: Shuo Zhang. Email: szhang18@gzhu.edu.cn

Received: 30 July 2020 Accepted: 15 September 2020

ABSTRACT

MapReduce is a widely used programming model for large-scale data processing. However, it still suffers from the skew problem, which refers to the case in which load is imbalanced among tasks. This problem can cause a small number of tasks to consume much more time than other tasks, thereby prolonging the total job completion time. Existing solutions to this problem commonly predict the loads of tasks and then rebalance the load among them. However, solutions of this kind often incur high performance overhead due to the load prediction and rebalancing. Moreover, existing solutions target the partitioning skew for reduce tasks, but cannot mitigate the computational skew for map tasks. Accordingly, in this paper, we present DynamicAdjust, a run-time dynamic resource adjustment technique for mitigating skew. Rather than rebalancing the load among tasks, DynamicAdjust monitors the run-time execution of tasks and dynamically increases resources for those tasks that require more computation. In so doing, DynamicAdjust can not only eliminate the overhead incurred by load prediction and rebalancing, but also culls both the partitioning skew and the computational skew. Experiments are conducted based on a 21-node real cluster using real-world datasets. The results show that DynamicAdjust can mitigate the negative impact of the skew and shorten the job completion time by up to 40.85%.

KEYWORDS

MapReduce; task scheduling; resource allocation; data skew; big data

1 Introduction

Over the past decade, MapReduce has been found to have tremendous practical value for big data processing. Many companies, including Facebook, Amazon, Twitter and Yahoo!, have used MapReduce to process data-intensive jobs on a daily basis. Although several different big data platforms, such as Hadoop, Hive, Spark, and Storm, have recently emerged one after another, these platforms are all essentially based on the fundamental concept of MapReduce. In MapReduce, there are two kinds of tasks, namely map tasks and reduce tasks. Each map task takes a single chunk of input data as input and outputs the intermediate results to the local disks. Subsequently, each reduce task fetches the intermediate results remotely and outputs the final results into the distributed file system. By breaking down a large data analytic job into a

number of smaller tasks and executing these tasks on multiple machines in a distributed manner, MapReduce can dramatically reduce the job completion time.

While MapReduce is highly successful, it remains affected by many issues. One of these, skew, is particularly challenging to eliminate; by "skew," we here refer to the case in which the load is imbalanced among tasks. Skew can make a small number of map or reduce tasks take significantly longer to finish than other tasks of the same kind, thereby increasing the job completion time. Let us take the PageRank application as an example, which is a link analysis algorithm that counts the number of incoming edges of each vertex using a graph. This application exhibits skew when the input graph includes nodes with far more incoming edges than others, which is very common in real workloads. Kwon et al. [1] have shown the existence of skew in PageRank using a real-world dataset. In more detail, the slowest map task takes more than twice compared to the second-slowest map task, with the latter still being five times slower than average. Since an individual job's running time is bounded by the time taken to complete the slowest task, skew can severely prolong the job completion time.

Many skew mitigation approaches for MapReduce have been proposed. The majority of these adopt a common approach that estimates the distribution of the intermediate key-value pairs and then reassigns these key-value pairs to tasks. However, for the purpose of predicting the key-value distribution, this will cause a synchronization barrier, as it requires either waiting for all map tasks to be completed [2,3] or adding a sampling procedure prior to the actual job beginning [4–7]. Some other approaches [8,9] have speculatively launched replica tasks for lagging tasks with the expectation that the former will be completed faster than the latter. Nevertheless, executing replica tasks for tasks affected by data skew cannot improve the job running time. Unlike the above methods, which reassign key-value pairs before the task execution, the approach proposed by Kwon et al. involves a real-time skew mitigation technique, called Skewtune [10], which repartitions the remaining workloads of lagging tasks and creates a new MapReduce job to process the remaining load. Nevertheless, this approach creates significant runtime overhead that extends to 30 s [10]. This overhead should not be underestimated even for short running tasks that only last 100 s, as this kind of task is very common in production workloads [11,12].

In light of the limitations of existing approaches, a run-time partitioning skew mitigation technique based on dynamic resource allocation has also been proposed [13,14]. This technique dynamically allocates resources in order to reduce tasks with reference to the estimated load of these tasks. The reduce tasks that take more work to process will be allocated more resources. Hence, this approach can decrease the variation in the running times of reduce tasks, thereby mitigating the negative impact of data skew. However, this mitigation solution focuses on the skew for reduce tasks; it cannot handle skew occurring in map tasks. As we can see in [1], skew is prevalent in the map stage in practice, which limits the applicability of this approach.

In light of the above, this paper presents DynamicAdjust, a run-time dynamic resource adjustment approach for skew mitigation in MapReduce. Unlike existing solutions, DynamicAdjust monitors the execution of tasks at runtime and adjusts resources for the slow-running tasks while there are still idle resources to draw on. Moreover, DynamicAdjust can mitigate the skew that arises in both the map and reduce stages. The contributions of this work can be summarized as follows:

— We develop a phase-aware remaining task time prediction technique, which considers the progress rate in each phase individually. Since each phase of a task runs at different speeds

in real-world workloads, this phase-aware task remaining time prediction technique can outperform existing prediction techniques that assume the progress rate is constant.
— We propose a resource adjustment algorithm that dynamically increases the size of containers for the slow running tasks at run-time. In this way, our approach can decrease the variation in task running time, thereby mitigating the skew.
— We evaluate DynamicAdjust through experiments based on a 21-node real cluster using real-world datasets. The experimental results show that DynamicAdjust can shorten the job completion time by up to 40.85%.

A preliminary version of this work was previously introduced in a letter in [15]. This paper extends our preliminary work in a number of respects. First, the skew detection algorithm and the phase-aware remaining task time prediction are presented here. Second, we have provided more details of the skew mitigation strategy. Third, additional experiments using the RelativeFrequency and Grep applications have been conducted to evaluate the effectiveness of DynamicAdjust.

The remainder of this paper is organized as follows. Section 2 describes the background and the motivation of our work. Section 3 presents the architecture of DynamicAdjust, while Section 4 illustrates the design of DynamicAdjust in detail. We provide the experimental evaluations in Section 5. Finally, the existing works related to DynamicAdjust are summarized in Section 6, after which our conclusions are presented in Section 7.

2 Background and Motivation

In this section, we discuss the two types of skew that arise in MapReduce and outline the resource management mechanism employed in a widely used MapReduce implementation (i.e., Hadoop YARN), which forms the motivation for this study.

2.1 Skew in MapReduce

Skew is a prevalent issue that can be encountered in many application domains, including database operations [16], scientific computing [1] and search engine algorithms [17]. The two kinds of skewness in MapReduce are summarized below:

— Computational Skew: This type of skew can occur in both the map and the reduce stages. In the map stage, each map task processes a sequence of records in the form of key-value pairs. Under ideal circumstances, these records have similar resource requirements (e.g., CPU and memory), resulting in the execution time being the same. In reality, however, there may be some expensive records that require more computation; this can cause variation in the task runtime, even though the size of the data chunk processed by each map task is the same. Similarly, in the reduce stage, some expensive key groups processed by the tasks can also skew the running time of the reduce tasks, regardless of the sizes of these key groups.
— Partitioning Skew: This type of skew arises most commonly in the reduce stage. In the original MapReduce systems, a hash function, *HashCode* (intermediate key) MOD number of reduce tasks, is used to distribute the intermediate data among reduce tasks [18]. This hash function works well when the keys are uniformly distributed; however, it may fail while processing non-uniform data. This may be caused by two factors. First, some keys appear with higher frequency than others in the intermediate data, causing the reducers that process these keys to become overloaded. Second, the sizes of values for some keys are significantly larger than others, even when the key frequencies are uniformed; thus, load imbalance among reduce tasks may still arise.

2.2 Resource Management in Hadoop YARN

Hadoop YARN [19] is the state-of-the-art implementation for MapReduce. The task scheduling functionalities are divided into two categories: *ResourceManager*, which is in charge of allocating resources to the running applications while satisfying constraints of capacity, fairness, etc., and *ApplicationMaster*, which is responsible for negotiating the acquisition of the appropriate resources from *ResourceManager* and allocating the resources to the tasks for each job. In addition, YARN uses containers to manage the resources in the cluster. In a given container, the amount of resources (e.g., ⟨2GB RAM; 1CPU⟩) can be customized for each task. However, YARN neglects to address the fact that resource requirements vary for different tasks, while also failing to provide dynamic resource adjustment for these tasks; hence, these tasks will require more resources (i.e., tasks take more work to process) and may run slower than other tasks because of the resource deficiency. Since the job completion time is determined by the finish time of the slowest task, the static resource scheduling mechanism in native YARN may prolong the job completion time in cases where the load of tasks is unevenly distributed.

Most existing works [2–4,20,21] only target the partitioning skew and neglect the computational skew that can arise in both the map and reduce stages. Moreover, a common approach is adopted to these solutions that predicts and then redistributes the task load to achieve a better balance, which requires additional (sometimes heavy) overhead in terms of key distribution sampling and load reassignment. In comparison, DynamicAdjust uses an alternative approach that dynamically increases the resources available to the tasks that require more computation; this not only effectively mitigates both the computational and partitioning skew, but also simultaneously eliminates the mitigation overhead. In the following, we will discuss the design of DynamicAdjust in detail.

3 System Architecture

Fig. 1 illustrates the architecture of DynamicAdjust. There are five main modules: the *SkewDetector*, the *SkewMitigator*, the *ResourceRequester*, the *ContainerAdjuster* and the *ResourceScheduler*. The *SkewDetector* is responsible for monitoring the progress of each task and detecting the skew at run-time. The *SkewMitigator* takes the tasks detected to have skew as input, calculates the amounts of resources that need to be adjusted and notifies the *ResourceRequester* to ask the *ResourceManager* for resources. The *ResourceRequester* then communicates with the *ResourceManager*; specifically, it is in charge of sending the resource adjustment requests to *ResourceManager* through heartbeat messages and handling the resource adjustment responses accordingly. The *ContainerAdjuster* talks to the *NodeManager* and adjusts the container size by calling the remote function in *NodeManager*. Finally, the *ResourceScheduler* allocates the cluster's resources based on scheduling policies such as fairness and capacity. It should be noted that current resource scheduling in native YARN cannot dynamically perform resource adjustment for a given task. We have therefore extended the resource scheduling to handle the resource adjustment requests.

The workflow of DynamicAdjust is as follows:

1. The *SkewDetector* gathers the task reports from the running tasks at runtime. After assessing the progress, the *SkewDetector* detects skewed tasks and then sends a notification to the *SkewMitigator*.
2. Upon receiving a notification that a skewed task has been detected, the *SkewMitigator* computes the amount of resources required for these tasks, then sends a notification

to the *ResourceRequester*. Subsequently, the resource adjustment request is sent by the *ResourceRequester* to the *ResourceScheduler*.

3. Once the resource adjustment request is received by the *ResourceScheduler*, it will schedule free resources to be sent to the *ResourceRequester* in the associated *ApplicationMaster*, based on the resource inventory in the cluster.
4. When positive responses are received by the *ResourceRequester*, it sends a notification to the *ContainerAdjuster*. The *ContainerAdjuster* then executes the resource adjustment with the cooperation of the *NodeManager*.

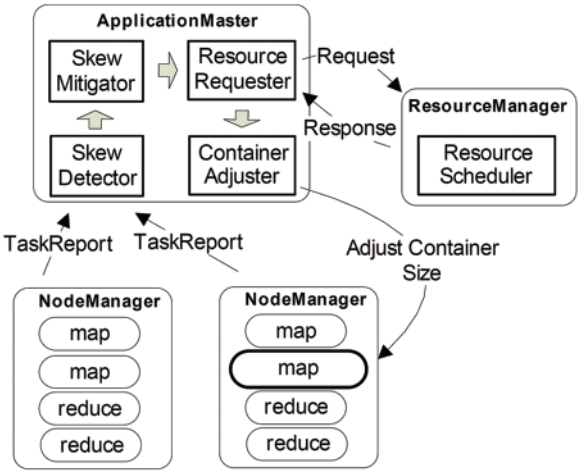

Figure 1: Architecture of dynamicadjust

4 DynamicAdjust Design

In this section, we describe the design of our proposed skew mitigation approach in detail. There are two main challenges in DynamicAdjust. First, in order to mitigate skew at run-time, it is necessary to develop an accurate skew detection technique. Second, after the skew has been detected, the next challenging step is to effectively mitigate the negative impact of skew without incurring noticeable overhead. We will provide our proposed solutions for these challenges in the following sections.

4.1 Skew Detection

It is nontrivial to identify skew before task execution while being agnostic to the characteristics of the data that a task is processing. Although each task's input data is of identical size, skew (e.g., computational skew) may also arise. Many existing solutions detect skew by gathering the information of key-value pairs or sampling the load distribution; however, these solutions either have to wait for all map tasks to be completed or perform sampling before actual jobs are run. Le et al. [22] have demonstrated that waiting for all map tasks to be completed will significantly increase the job completion time. A run-time skew detection technique is therefore desirable for skew mitigation.

Algorithm 1 Skew Detection Algorithm

Input: θ_{late}: threshold for waiting for sufficient historical statistics of completed tasks;
θ_{slow}: threshold for slow tasks.
Output: $task^{skew}$: a task to mitigate
1: $task^{skew} = null$
2: **if** $numWait \neq 0$ **and** $numComp < \theta_{late}\ \ numTotal$ **then**
3: **return**
4: **end if**
5: **for** $task$ in $Task^{running}$ **do**
6: $taskDurt = getTaskEstimatedDurt(task)$
7: **if** $\frac{taskDurt - avgTaskDurt}{avgTaskDurt} > \theta_{slow}$ **then**
8: $C = C \cup \{task\}$
9: **end if**
10: **end for**
11: **for** $task$ in C **do**
12: $remTime_{task} = getRemainingTime(task)$
13: **if** $remTime > remTime_{max}$ **then**
14: $task^{skew} = task$
15: **end if**
16: **end for**
17: **return** $task^{skew}$

4.1.1 Identifying Skewed Tasks

We consider tasks with a large workload to be skewed tasks, on which DynamicAdjust will execute the skew mitigation algorithm (further details are provided in Section 4.2). To achieve the goal of identifying skewed tasks, we need to answer the following two questions: (1) Which tasks should be labeled as skewed tasks? (2) When to detect skewed tasks thus activating the skew mitigation.

Notably, one of the most significant characteristics of skewed tasks is that they take more time to complete than other tasks [23]. Hence, similar to the existing work in [23], we determine a task to be a skewed task candidate when the task's estimated running time exceeds the historical average task running time by θ_{slow} percent. Here, the historical average task running time is the average of the durations of the completed tasks. We introduce a threshold, θ_{late}, such that we will not initiate the skew detection algorithm until θ_{late} percent of tasks have been completed. As a result, the task durations for the completed tasks can be used as historical data[1]. Moreover, performing mitigation for skewed tasks that are near completion will be minimally beneficial, even though they have a longer run-time. Therefore, we select one task from the skewed task candidates, i.e., the task with the longest remaining time at the moment of detection. We will describe how the remaining and running time of a task is predicted in Section 4.1.2.

The next question is that of when skewed tasks should be detected. In DynamicAdjust, we will request resources for skew mitigation once a task is identified as skewed. As a result, if we begin the skew detection too early, there may still be new tasks waiting to be executed at that

[1] Note that using historical statistics of the completed tasks to improve the scheduling in MapReduce is common [24,25].

time. Thus, requesting resources for skewed tasks will result in resource competition with the new tasks, thereby postponing their executions. If we start the skew detection too late, moreover, DynamicAdjust may miss the right time to mitigate the skew, meaning that no benefit will be gained. Similar to existing speculation mechanisms in Hadoop [8], we begin the skew detection when there is no task waiting for resources.

Algorithm 1 outlines the skew detection strategy in detail. Note that map-side skew and reduce-side skew are treated separately, but the strategy in Algorithm 1 is applicable to both of these cases. As shown in lines 2–4 of Algorithm 1, DynamicAdjust waits until no task is requesting the container and θ_{late} percent of tasks have been completed before evaluating the tasks for skew mitigation. It then iterates the running tasks to obtain their estimated task running time (Line 5–10). If the estimated task running time exceeds the average task duration by θ_{slow} percent, this task will be labeled as a skewed task candidate. After that, DynamicAdjust selects the candidate with the longest remaining time as the task most likely to be skewed at this time (lines 11–16). The time complexity of this algorithm is O(n), where n represents the number of map tasks.

4.1.2 Phase-Aware Task Remaining Time Prediction

In order to monitor the task progress, Hadoop arranges a fixed percentage of progress to each phase of a task [8,9]: For example, 66.7% for the map phase and 33.3% for the merge phase in a map task, as well as 33.3% for each of the three phases (i.e., copy, sort, and reduce) in a reduce task. Moreover, in each phase, Hadoop monitors the fraction of data processed and sets the corresponding score in this phase accordingly. Hence, for a reduce task that is processed halfway through the sort phase, its progress score is $33.3\% + 0.5 \times 33.3\% = 49.95\%$. The state-of-the-art solution, LATE, simplifies the estimation of the time remaining for tasks by assuming that the progress rate is stable in each phase. The remaining task time is then calculated by $\frac{1-progressScore}{progressRate}$, where progressRate $= \frac{progressScore}{Time_{elapsed}}$ [8]. However, the duration of each phase is not necessarily proportional to its progress percentage; as a result, the remaining task time may be incorrectly predicted. Fig. 2 presents the average duration of each phase while running PageRank and InvtIndex. We can observe that the durations of the map phases are 70.62% and 94.31% of the total task durations for InvtIndex and PageRank respectively; this contradicts the assumption made by LATE. As shown in Fig. 2b, the same result can be obtained in the reduce stage. Furthermore, Fig. 3 compares the progress rate of each phase between PageRank and InvtIndex. It can be seen that there are significant differences in the progress rate for different phases of a task. Hence, LATE's estimation of the time remaining for a task based on an unchanged progress rate is problematic.

There are two potential ways to eliminate this problem: (1) Adjust the percentage of the task progress for each phase and make it proportional to the duration of the corresponding phase; (2) Use different progress rates in different phases. Both of these approaches treat each phase separately and target the same goal. To keep our algorithm simple, we opt for the latter approach, which only requires modifying the remaining time calculation logic in *ApplicationMaster* and continues to use the progress score provided by Hadoop. More specifically, we predict the remaining time of a task based on the sum of the remaining time left in the current and subsequent phases. Here, we calculate the time remaining for a task in the current phase based on the progress rate measured in this phase, then calculate the remaining time for the following phases based on the average progress rates in the corresponding phases of the completed tasks. Note that we record the historical data and the average progress rate per phase so to improve the

ease of calculation. Accordingly, the phase-aware task remaining time $T_{task}^{remaining}$ can be calculated as follows:

$$T_{task}^{remaining} = T_{cur}^{remaining} + \sum T_{fol}^{duration} \tag{1}$$

$$= \frac{Pcent_{cur}^{remaining}}{progressRate} + \sum AvgDurt_{fol} \cdot Factor,$$

$$Pent_{cur}^{remaining} = Pcent_{cur} - (ProScore - \sum Pcent_{com}),$$

$$progressRate_{cur} = \frac{ProScore - \sum Pcent_{com}}{T_{cur}^{elapsed}},$$

$$Factor = \frac{T_{cur}^{duration}}{AvgDurt_{cur}} = \frac{T_{cur}^{remaining} + T_{cur}^{elapsed}}{AvgDurt_{cur}},$$

where $Pcent_{cur}^{remaining}$ and progressRatecur respectively denote the remaining percentage still needing to be executed and the estimated progress rate of the current phase, while Pcentcur, Pcentcom and Pcentfol are the percentages of task progress for the current, completed and following phases, respectively (e.g., 33.3 percent for the Copy phase by default). Moreover, ProScore is the task progress score provided by Hadoop, $T_{com}^{duration}$ is the duration of the completed phase, $T_{cur}^{elapsed}$ is the time elapsed in the current phase, $T_{task}^{elapsed}$ is the time elapsed for the task in question, and AvgDurtcur and AvgDurtfol represent the average durations of the current and following phases, respectively. We also introduce a scaling factor, Factor, to scale up the estimated durations of the following phases with reference to the ratio between the estimated duration of the current phase and the historical average duration of the current phase.

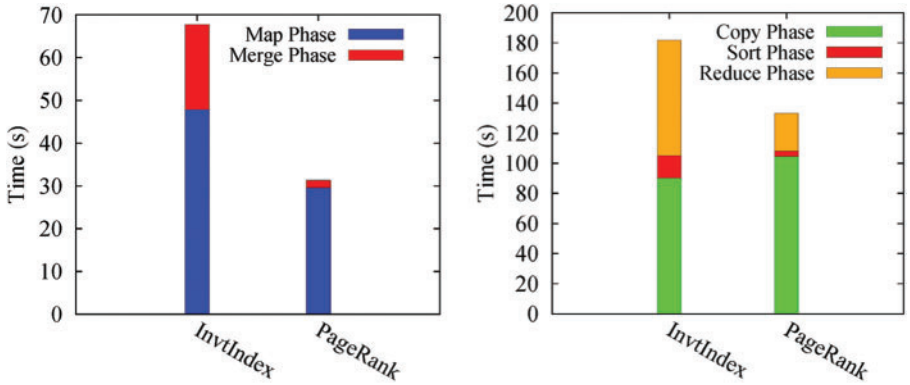

Figure 2: Average duration of each phase (a) map stage (b) reduce stage

Accordingly, the task running time $T_{task}^{duration}$ can be calculated as follows:

$$T_{task}^{duration} = T_{task}^{elapsed} + T_{task}^{remaining} \tag{2}$$

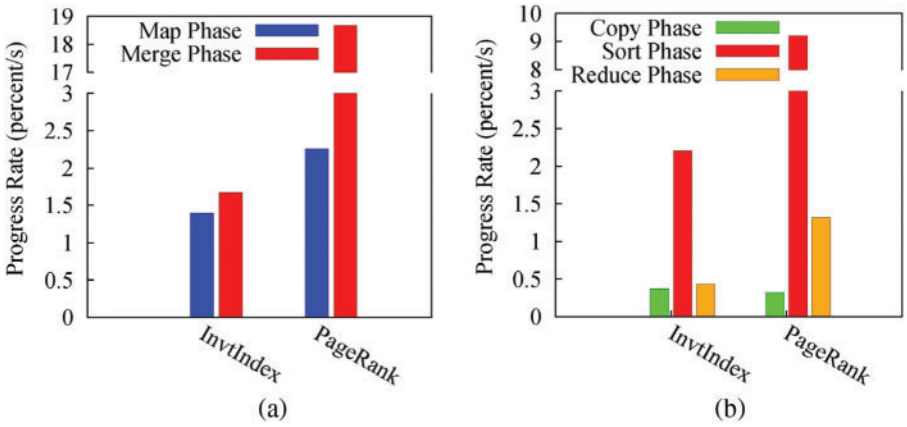

Figure 3: Progress rate of each phase (a) map stage (b) reduce stage

To illustrate how the phase-aware task remaining time prediction works, we take a reduce task of a PageRank job as an example. There are three phases of a reduce task: Namely, Copy, Sort and Reduce. These different phases run at different rates, as shown in Fig. 2b. Suppose that a reduce task has completed the Copy phase at 102 s and is now in the Sort phase with a progress score of 50 at 105 s. According to LATE, the remaining time of this task is 105 s and the estimated task completion time is 210 s. However, the progress rate in the Sort phase is obviously faster than in the Copy phase. By contrast, our approach predicts the remaining time based on the estimated progress rate of the current phase $progressRate_{cur}$ and the average duration of the following phase (i.e., the Reduce phase) $AvgDurt_{fol}$. Suppose that we obtain $AvgDurt_{fol} = 22$ s from the historical profile and set $Factor = 1$. We can thus obtain $T_{task}^{remaining} = \frac{34\%-(50\%-33\%)}{50\%-33\%} \times (105 - 102) + 22 = 25$ s and $T_{task}^{duration} = 130$ s. Fig. 4 illustrates the prediction logic of LATE and DynamicAdjust for this example. From the figure, we can observe that the prediction of DynamicAdjust is closer to the average task duration 132 s (see Fig. 2b).

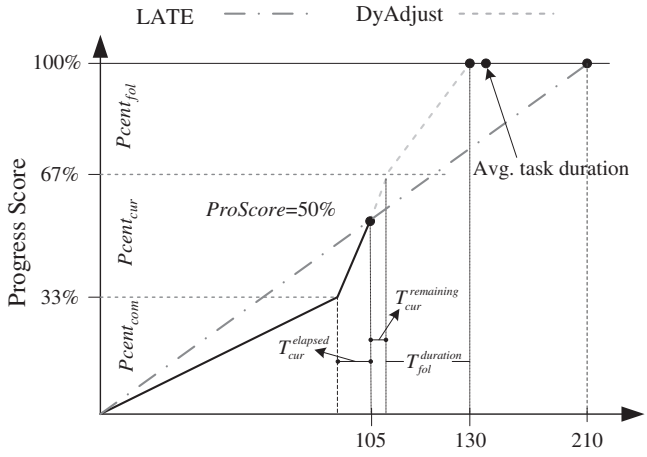

Figure 4: An example of remaining time prediction for LATE and DynamicAdjust

4.2 Skew Mitigation

After the skewed tasks have been detected, we next need to mitigate them in order to reduce the job completion time. Unlike existing solutions, which repartition the workload of skewed tasks, we opt to dynamically adjust the resources for these tasks. To do this, it is necessary to determine the impact of resource allocation on task duration. In the following sections, we will describe our skew mitigation approach in detail.

4.2.1 Impact of Resources on Task Running Time

As Hadoop YARN only supports allocation for two types of resources (i.e., CPU and memory), we here concentrate on understanding the impact of CPU and memory allocation on the task completion time.

We perform a series of experiments on our Testbed cluster (see Section 5 for more details) by varying the resource allocation for the tasks. More specifically, with respect to the impact of the CPU, we record the task durations by changing the CPU allocation for this task from 1 *vCore* to 8 *vCores* and fixing the memory allocation at 1 *GB*. It is notable here that the reason why we fix the memory allocation to 1 *GB* is to isolate the impact of the CPU allocation. Fig. 5 illustrates the impact of the CPU for a task that has been randomly selected from InvtIndex. It can be clearly seen from the figure that the task duration decreases sharply in the beginning as the CPU allocation increases. When the CPU allocation exceeds a certain threshold (i.e., 3 *vCores* in this experiment), small changes in the task duration can be observed. Accordingly, the non-linear regression method is used to model the relationship between CPU allocation and the task duration by the following equation:

$$T_{task} = \alpha_1 + \frac{\beta_1}{Alloc^{cpu}} \tag{3}$$

where T_{task} is the task running time and Alloc^{cpu} is the CPU allocation to the task. The regression result is indicated in Fig. 5 by the green line. We also found that the fitting result is inaccurate when CPU allocation is large; this is because, as the CPU allocation gradually increases, the resource bottleneck may switch from the CPU to other types of resources (e.g., network bandwidth), meaning that the benefits of additional CPU allocation to task durations would decrease. A similar observation is also made in [26], in which augmenting the network bandwidth beyond a certain threshold was found to have no impact on job completion time when the job is restricted by disk performance. We then use a piecewise inversely proportional model to remedy this regression error. As can be seen from in Fig. 5, the red line fits better than the green line.

Similarly, we perform a set of experiments by varying the tasks' memory allocation. The CPU resources allocated to a task are determined by the number of *vCores* allocated to the task. Memory allocation, by contrast, is controlled by two configurations: Logical RAM limit and maximum JVM heap size limit[2]. The former is a unit used to manage the resources logically, while the latter setting reflects the maximum heap size of the JVM that runs the task. Hence, the maximum JVM heap size limit is the actual controller that determines the maximum usable memory for the task. Consequently, we record the task durations by varying the JVM heap size limit from 200 MB (the default value) to 4000 MB and maintaining the CPU allocation at 1 vCore. Similarly, we fix the CPU allocation to 1 vCore in order to isolate the impact of memory

[2] The logical RAM limit and maximum JVM heap size for a map task correspond to *mapreduce.map.memory.mb* and *mapreduce.map.java.opts* respectively in the Hadoop configuration file *mapred-default.xml* [27].

allocation. We further use the non-linear regression method to ascertain the relationship between task duration and memory allocation by following the below inverse proportional model:

$$T_{task} = \alpha_2 + \frac{\beta_2}{Alloc^{mem}} \tag{4}$$

where T_{task} is the task running time and $Alloc^{mem}$ is the memory allocated to the task. Fig. 6 plots the impact of memory allocation for a task randomly selected from InvtIndex. It is clear that the inverse proportional model is also applicable to the relationship between task duration and memory allocation. In more detail, task duration decreases dramatically at the beginning; after the memory allocation becomes sufficient for the task, minimal improvement can be obtained by continuing to increase the memory allocation. However, with respect to the map task, no change in the task duration can be observed while increasing the memory allocation. This is because each map task processes one split of the data chunk in HDFS, and the size of each data chunk is 128 MB. The minimum JVM heap size (200 MB) is already sufficient for the map task.

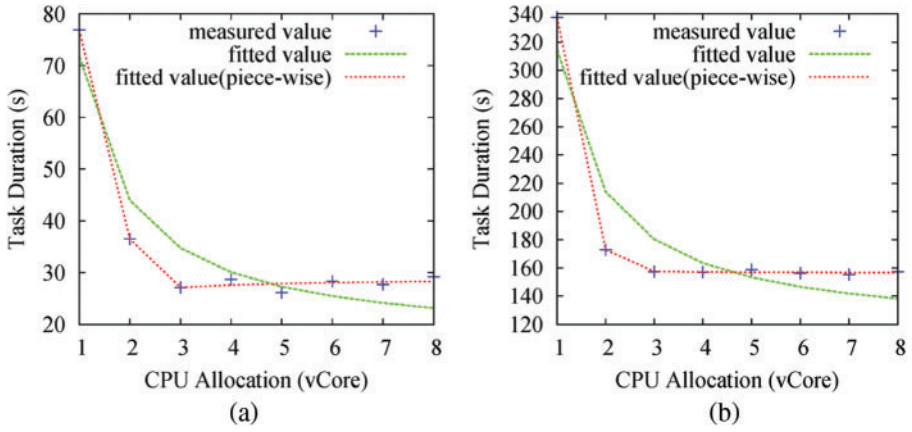

Figure 5: Relationship between the task duration and the CPU allocation in InvtIndex (a) a map task (b) a reduce task

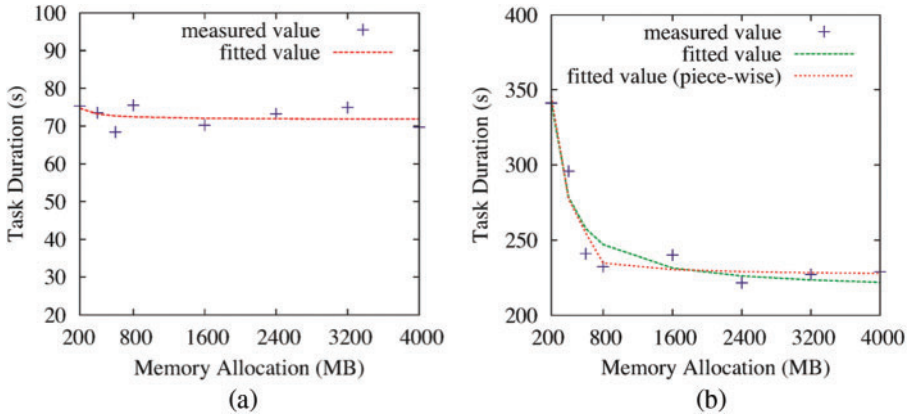

Figure 6: Relationship between task duration and memory allocation in InvtIndex (a) a map task (b) a reduce task

4.2.2 Skew Mitigation Strategy

Since the native Hadoop scheduler allocates an identical amount of resources to each task of the same type (map or reduce task), tasks that require more computation will run for longer than other tasks due to the resource bounding, which will significantly prolong the job completion time. On the other hand, we have demonstrated in the previous section that the impact of resource allocation on the duration can be modeled as an inverse proportional model. We accordingly adopt a simple strategy that dynamically adjusts resources for the detected skewed tasks in a way that accelerates their task running times, thereby mitigating the negative impact of the skew.

More specifically, we increase the CPU allocation to $\frac{T_{task}^{durt}}{AvgTaskDurt} Alloc_{old}$ for the skewed tasks, where $Alloc_{old}$ is the original CPU allocation of the task. Moreover, DynamicAdjust does not perform the memory adjustment at run-time; this is because YARN uses the JVM-based container to execute a task, and the current technique does not support dynamically changing the JVM heap size for a container. For simplicity, DynamicAdjust sets the JVM heap size for each task to 80% of the logical memory limit when the task is first launched. The rationale behind this is that we manage to maximize the memory under the condition that the resource bounding is also obeyed.

In short, this resource allocation strategy is sample but works well in practice. It roughly predicts the amount of resources required for the skewed tasks to run as fast as the normal tasks, then gives preferential treatment to these tasks by increasing their resource allocation accordingly. Even though the memory allocation is not changed dynamically at run-time, we increase the default setting of the JVM heap size before the tasks are run. We further observed that 800 MB[3] is in fact a sufficient JVM heap size for a task during our experiments.

Admittedly, DynamicAdjust also incurs overhead, specifically the amount of additional resources allocated to the skewed tasks, since DynamicAdjust scales up the CPU resource allocation for skewed tasks in order to accelerate their running times. However, the number of skewed tasks is small compared to the number of tasks in total; as a result, the amount of additional resources incurred by DynamicAdjust is also small.

5 Evaluation

Our experiments are conducted on a Hadoop cluster with 21 nodes. Each node has four Genuine Intel 2 GHz processors, 8 GB RAM and an 80 GB high-speed hard drive. We deploy *ResourceManager* and *NameNode* on one node, while the other 20 nodes are deployed as workers. Each worker is configured with eight virtual cores and 7 GB RAM (leaving 1 GB for background processes). The minimum CPU and memory allocations for each container are 1 vCore and 1 GB respectively. The HDFS block size is set to 128 MB and the replication level is set to 3. The configuration of Cgroups and the output compression are enabled.

The following MapReduce applications are used in the evaluation:

— InvtIndex: The inverted index data structure is widely used in the search engine indexing algorithm. This application is implemented in Hadoop and builds the inverted index for a text dataset. We use a 30 GB Wikipedia dataset in the evaluation.

[3] The logical RAM limit is 1 GB and the JVM heap size limit is 400 MB by default. Eighty percent of the minimum logical RAM limit is 800 MB.

— RelativeFrequency: This measures the proportion of time for which word w_j co-occurs with word w_i within a specific range, which can be denoted as $F(w_j|w_i)$. We adopt the implementation provided in [28] and use a 30 GB Wikipedia dataset as input.
— PageRank: This is a link analysis algorithm that estimates the importance of the website. It assigns a rank to each page (vertex) according to the structure of the hyperlinks (edges) connected to it. We adopt the implementation provided in [29] and use the web crawl as the input dataset.
— Grep: This application is a benchmark included in the Hadoop distribution, which extracts matching strings from text files and counts how many times they occur. In the evaluation, we grep "computer" from a 30 GB Wikipedia dataset.

5.1 Accuracy of Skew Detection

To evaluate the accuracy of our skew detection technique, we run MapReduce jobs with only the skew detection activated and without performing the skew mitigation operation. Accordingly, after the skewed tasks are detected, we label the tasks and let them run normally. At the end, we verify whether these tasks are truly skewed tasks. Consistent with [9], skewed tasks are determined by the task durations that exceed the average by 50%. Moreover, we perform the experiments in the same way as in Hadoop-LATE (the implementation of LATE in Hadoop), the state-of-the-art skew detection solution. Fig. 7 presents the comparison of the skew detection precision[4] between DynamicAdjust and Hadoop- LATE.

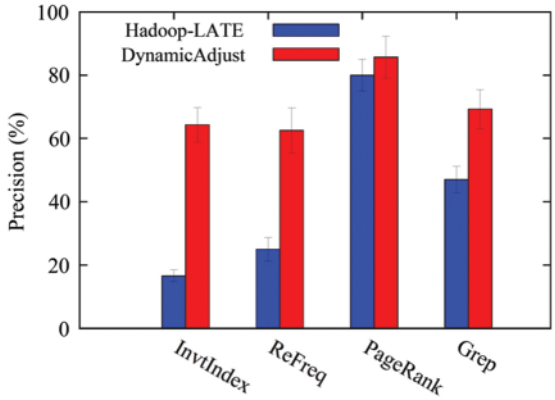

Figure 7: The skew detection accuracy between DynamicAdjust and Hadoop-LATE

It is clear from the figure that DynamicAdjust has significantly higher precision than Hadoop-LATE. More specifically, DynamicAdjust improves the precision by 47.61%, 37.5%, 5.7% and 22.17% for InvtIndex, RelativeFrequency, PageRank and Grep, respectively.

To facilitate understanding of the reasons why DynamicAdjust is superior, the skew detection statistics of InvtIndex are illustrated in detail in Tab. 1. It is clear from the table that in the map stage, all the truly skewed tasks can be identified by both methods (i.e., Hadoop LATE and DynamicAdjust). However, the False Positive values for Hadoop-LATE and DynamicAdjust are 25 and 5, respectively; this indicates that 25 normal tasks are detected as skew tasks by Hadoop-LATE, whereas DynamicAdjust misjudges five tasks as skewed ones. For the reduce tasks,

[4] The precision is calculated by $\frac{TruePositive}{TruePositive+FalseNegative}$.

moreover, none of the skewed tasks can be detected by Hadoop-LATE; by contrast, DynamicAdjust detects four out of five skewed tasks. Hadoop-LATE has low precision because it assumes the progress speed of each phase remains unchanged; however, this is not true, as demonstrated in Section 4.1.

Table 1: Skew detection details of InvtIndex (the number of map and reduce tasks in the experiments are 235 and 60, respectively. TP: True Positive; TN: True Negative; FP: False Positive; FN: False Negative.)

Strategy	Map stage				Reduce stage			
	TP	TN	FP	FN	TP	TN	FP	FN
Hadoop-LATE	5	205	25	0	0	55	0	5
Dynamic-Adjust	5	225	5	0	4	55	0	1

5.2 Performance of the Skew Mitigation

To evaluate the performance of the skew mitigation for DynamicAdjust, we compare it against the following methods: (1) Native Hadoop YARN; (2) A speculation-based straggler mitigation method (Hadoop-LATE); (3) A repartition-based skew mitigation method (SkewTune) and (4) A resource allocation-based skew mitigation method (DREAMS [13]). Since SkewTune is deployed on Hadoop v1, it is slot-based and provides no resource isolation for the slots. To facilitate fair comparison of the above methods, we add the resource isolation between slots in Hadoop v1 (0.21.0) and implement SkewTune on it. Moreover, each node is configured with six map slots and two reduce slots while using SkewTune. Fig. 8 compares the job completion time for different MapReduce applications while running different mitigation methods. It can be seen from the figure that DynamicAdjust performs better than other skew mitigation methods in all cases. More specifically, DynamicAdjust reduces the job completion time by 32.86%, 40.85%, 37.27% and 22.53% for InvtIndex, RelativeFrequency, PageRank and Grep, respectively. Furthermore, no noticeable improvement can be observed in the experiments for Hadoop-LATE and SkewTune, which perform worse than YARN in some cases; for example, SkewTune consumes more time than native YARN in the PageRank application, as shown in Fig. 8c. In addition, DREAMS improves the job completion time for InvtIndex and RelativeFrequency, but yields no improvement for PageRank and Grep, which exhibit skew in the map stage.

To better explain the reason for the superiority of DynamicAdjust over other skew mitigation methods, we here illustrate the detailed execution timeline for different skew mitigation methods while running PageRank. As shown in Fig. 9a, for native YARN, the durations are significantly longer for some tasks than others in the map stage. Since the reduce function can only be performed after all intermediate data have been transferred, these slow-running map tasks delay the reduce phase start time. It is clear from Fig. 9a that a large amount of time (from 100 s to 200 s) is wasted due to the need to wait for the slow-running map tasks. Different from the map stage, all tasks have similar durations in the reduce stage.

Fig. 9b presents the execution timeline for Hadoop-LATE, which replicates the execution of these slow-running tasks by leveraging idle containers. This approach may accelerate these tasks; nevertheless, since these slow-running tasks process a skewed workload, while replicating these

tasks elsewhere will not alter the workload they need to process, the improvement obtained is not significant. As shown in Fig. 9b, the slow-running tasks still cannot be accelerated.

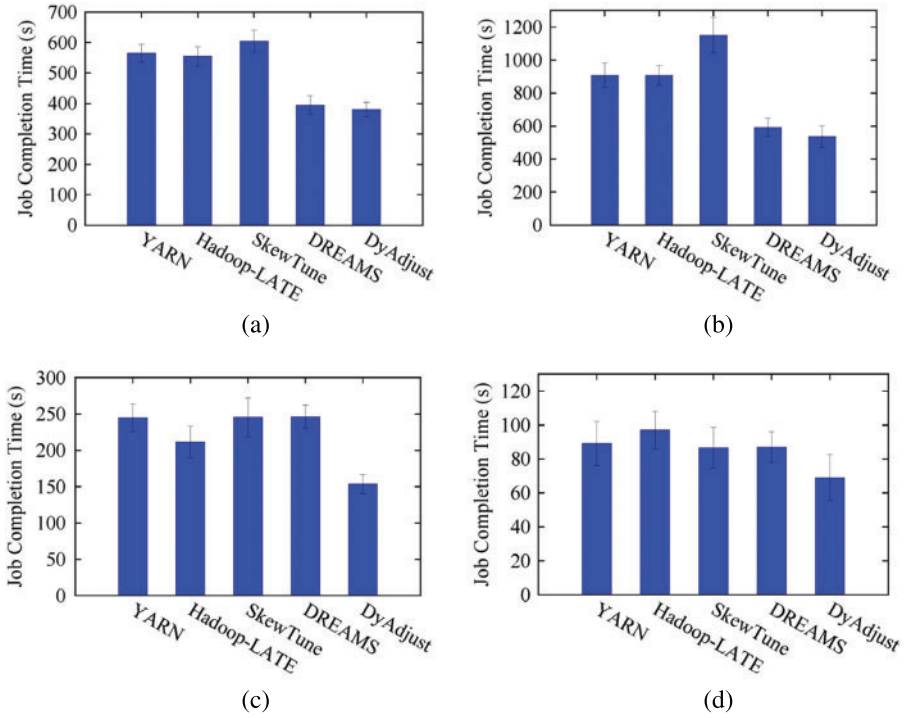

Figure 8: Job completion time comparison (a) InvtIndex (b) RelativeFrequency (c) PageRank (d) Grep

In comparison, SkewTune splits UP the unprocessed work of a skewed task and launches a new job (called the mitigation job) to process the incomplete work. This skewed task can consequently be accelerated through the use of free cluster resources. As can be seen from Fig. 9c, one of the slow-running map tasks is processed by a mitigation job and is finished earlier than the other skewed tasks. However, SkewTune can only mitigate one task at a time; in other words, if there is a mitigation job in progress, no new mitigation jobs can be launched (see Section 3.2 in [10]). As a result, if several skewed tasks arise concurrently, SkewTune may miss the best time to mitigate all skew. Fig. 9c presents one such example. Note that this is a real MapReduce application running on real-world data (ClueWeb09 web crawl [29]). As Fig. 9c shows, after the mitigation job is completed, no more tasks can be selected as skewed tasks; this is because SkewTune only selects the task with remaining time that is two times greater than the repartitioning overhead w (as reported in [10], $w = 30$ s). Therefore, SkewTune cannot improve the job completion time in this case.

Fig. 9d presents the execution timeline for DREAMS. As the figure shows, there is no improvement over native YARN for PageRank. This is because DREAMS offers no help to alleviate the skew in the map stage. This approach adjusts the resources for reduce tasks based on the estimated partition sizes; unfortunately, in PageRank, there is no data skew in the reduce stage. As shown in Fig. 9d, each reduce task has a similar duration, meaning that there is no room left for DREAMS to improve.

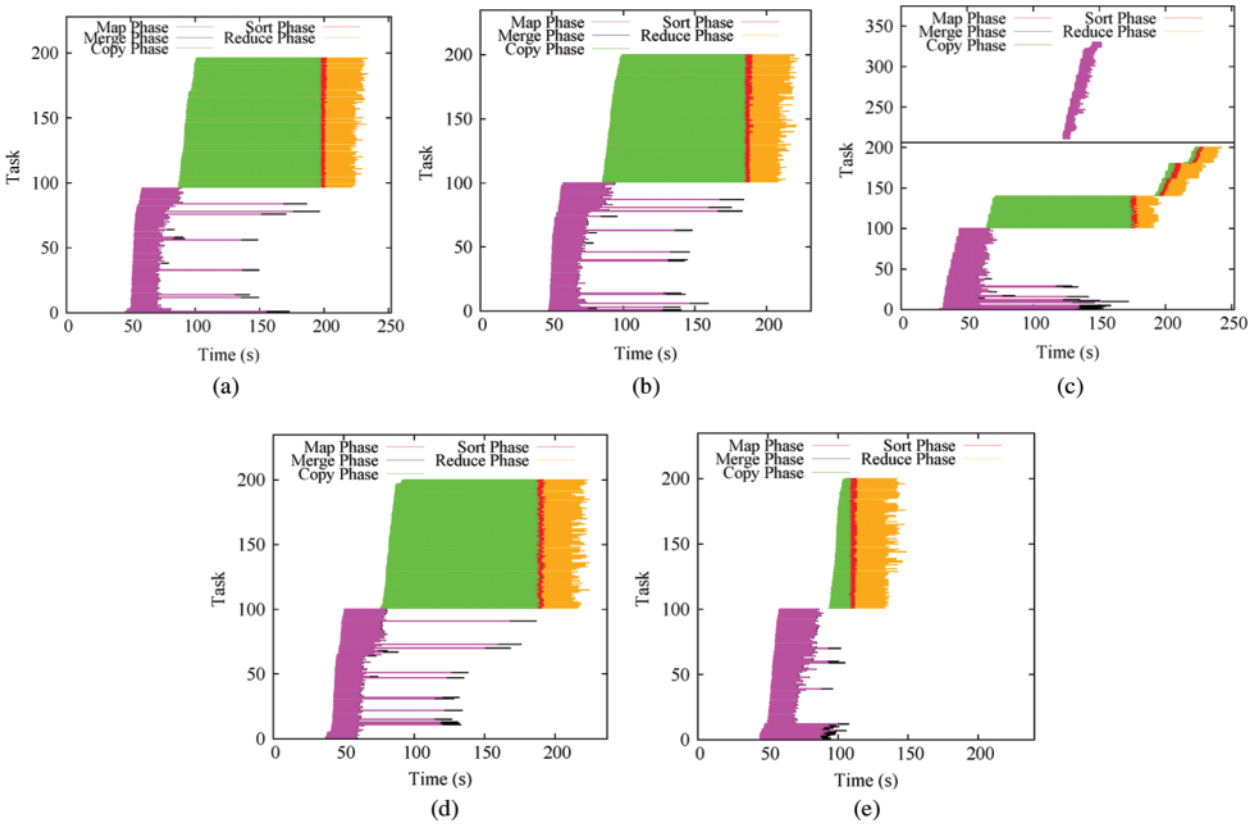

Figure 9: Job execution timeline comparison (a) YARN (b) Hadoop-LATE (c) SkewTune (d) DREAMS (e) DynamicAdjust

Similar to DREAMS, DynamicAdjust is a resource-aware skew mitigation technique. However, DynamicAdjust mitigates the skew in both the map and reduce stages. This approach detects the skew based on the estimated remaining task time and adjusts the resources for skewed tasks at run-time. It is clear from Fig. 9e that the skewed map tasks' durations are decreased significantly; as a result, the job completion time can be dramatically improved.

6 Related Work

Data skew for MapReduce. This problem has recently triggered widespread concern in the area of large-scale processing. In terms of skew detection, Chen et al. [9] consider a task to be a skew task when the predicted task durations exceed the average by 50%.

In terms of skew mitigation, Gulfer et al. proposed a load-balancing algorithm that reassigns the partitions based on a cost model. However, this kind of approach necessitates waiting for the completion of all map tasks before the reduce tasks can be launched; as shown in [30], this significantly prolongs the job. A progressive sampler [5] is proposed to predict the key distribution of the intermediate data. Rather than reassigning the large partitions before launching the tasks, SkewTune [10] repartitions the skewed partitions at run-time. However, this incurs an overhead associated with repartitioning the data and concatenating the output. In addition to the above solutions, Irandoost et al. [31] propose the traffic cost-aware partitioner (TCAP) to handle reducer-side data skewness in MapReduce. This approach attempts to balance the cost of network traffic

during shuffling while balancing the reducer load; nevertheless, this approach only targets data skewness in the reduce stage.

Stragglers in MapReduce. Dean et al. [18] first identified the stragglers problem in MapReduce in, in which these authors back up the running of pending tasks as the job nears completion. LATE [8] extends this work by speculatively launching a backup task for those slow-running tasks. However, running a replica task for a data-skew task (which has more data to process) may have a counter-productive impact. This is because duplicating the execution of the overload tasks cannot reduce the workload for these tasks. Unlike the work of [18], Mantri [23] culls stragglers based on their causes and executes tasks in descending order according to their input sizes. Nevertheless, Mantri assumes that the task workloads are obtained before a stage begins.

Resource-aware scheduling. As the application of large data processing to the IoT [32–34] has become increasingly common, the desire for fine-grained resource scheduling has become widespread. The Hadoop version 1 (MRv1) uses a slot-based resource allocation scheme and neglects the run-time task resource consumption while allocating the resources. Many solutions have accordingly been proposed to better utilize the resources in the cluster. For instance, Polo et al. [35] proposed a resource-aware scheduler, named RAS, which uses specific slots for scheduling. However, RAS places reduce tasks before map tasks; unfortunately, even when reduce tasks are allocated slots, they cannot begin until the map tasks have generated the intermediate data, meaning that the resources occupied by waiting reduce tasks are wasted. Therefore, Hadoop YARN [19] is proposed, which supports customizing the container size in the same way as specifying the amount of resources. However, YARN considers the resource consumption of the map task and the reduce task to be the same, which is problematic for skewed jobs. MROrchestrator, proposed by Sharma et al. [36], identifies the resource deficit based on the resource profiles and dynamically adjusts the resource allocation. Nevertheless, monitoring the resource consumption of each task at run-time is difficult to implement, since there are many background processes in the server that dynamically affect resource usage. Compared with MROrchestrator, our solution detects the skew task based on the remaining task time prediction, which is a simpler and more efficient approach. Moreover, there are other categories of resource scheduling policies in MapReduce, such as the work in [37–39]. These approaches schedule the resource in terms of the number of task slots in an attempt to achieve improved fairness or better resource utilization. However, these methods do not attempt to address data skew.

7 Conclusion

In this paper, we have presented DynamicAdjust, a run-time skew mitigation method. Unlike existing solutions, which estimate and then rebalance the loads of tasks, DynamicAdjust instead adjusts the resources at run-time to account for tasks that require more computation. This can completely eliminate the overhead caused by the load estimation and rebalancing. Compared to our previous work, DREAMS, DynamicAdjust makes no assumption regarding the causes of the skew, but instead monitors the progress of tasks; this enables it to mitigate both the partitioning skew and the computational skew that arise in the map or reduce stages. Finally, we conducted experiments based on real MapReduce workloads with a 21-node Hadoop cluster. The results demonstrated that DynamicAdjust increases the skew detection precision by up to 47.64% when compared to a baseline method named Hadoop-LATE. The results also revealed that DynamicAdjust can effectively mitigate the negative impact of skew and reduce the job completion time by up to 40.85% when compared to native YARN. Furthermore, existing solutions such as Hadoop-LATE and SkewTune are found to provide no improvement in our experiments, while

DREAMS cannot bring any benefit for PageRank and Grep applications that have skew in the map stage.

Acknowledgement: We thank Aimal Khan, Peixin Chen and Qi Zhang for very useful discussions.

Funding Statement: This work was funded by the Key Area Research and Development Program of Guangdong Province (2019B010137005) and the National Natural Science Foundation of China (61906209).

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

1. Kwon, Y., Balazinska, M., Howe, B., Rolia, J. (2011). A study of skew in mapreduce applications. *Open Cirrus Summit, 11(8),* 1–8.
2. Gufler, B., Augsten, N., Reiser, A., Kemper, A. (2011). Handing data skew in mapreduce. *Proceedings of the 1st International Conference on Cloud Computing and Services Science,* vol. 146, pp. 574–583.
3. Ouyang, X., Zhou, H., Clement, S., Townend, P., Xu, J. (2017). Mitigate data skew caused stragglers through imkp partition in mapreduce. *Proceedings of IEEE 36th International Performance Computing and Communications Conference,* pp. 1–8, San Diego, USA.
4. Tang, Z., Lv, W., Li, K., Li, K. (2018). An intermediate data partition algorithm for skew mitigation in spark computing environment. *IEEE Transactions on Cloud Computing,* 1–12. DOI 10.1109/TCC.2018.2878838.
5. Ramakrishnan, S. R., Swart, G., Urmanov, A. (2012). Balancing reducer skew in mapreduce workloads using progressive sampling. *Proceedings of the 3rd ACM Symposium on Cloud Computing,* pp. 1–14, San Jose, USA.
6. Yan, W., Xue, Y., Malin, B. (2013). Scalable and robust key group size estimation for reducer load balancing in mapreduce. *Proceedings of 2013 IEEE International Conference on Big Data,* pp. 156–162, Santa Clara, USA.
7. Gavagsaz, E., Rezaee, A., Haj Seyyed Javadi, H. (2018). Load balancing in reducers for skewed data in mapreduce systems by using scalable simple random sampling. *Journal of Supercomputing, 74(7),* 3415–3440. DOI 10.1007/s11227-018-2391-9.
8. Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. H., Stoica, I. (2008). Improving mapreduce performance in heterogeneous environments. *Proceedings of 2008 OSDI,* pp. 29–42, San Diego, USA.
9. Chen, Q., Liu, C., Xiao, Z. (2013). Improving mapreduce performance using smart speculative execution strategy. *IEEE Transactions on Computers, 63(4),* 954–967. DOI 10.1109/TC.2013.15.
10. Kwon, Y., Balazinska, M., Howe, B., Rolia, J. (2012). Skewtune: Mitigating skew in mapreduce applications. *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data,* pp. 25–36, Scottsdale, USA.
11. Alam, M., Shakil, K. A., Sethi, S. (2016). Analysis and clustering of workload in google cluster trace based on resource usage. *Proceedings of 2016 IEEE Intl Conference on Computational Science and Engineering and IEEE Intl Conference on Embedded and Ubiquitous Computing and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering,* pp. 740–747, Paris, France.
12. Mishra, A. K., Hellerstein, J. L., Cirne, W., Das, C. R. (2010). Towards characterizing cloud backend workloads: Insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review, 37(4),* 34–41. DOI 10.1145/1773394.1773400.
13. Liu, Z., Zhang, Q., Ahmed, R., Boutaba, R., Liu, Y. et al. (2016). Dynamic resource allocation for mapreduce with partitioning skew. *IEEE Transactions on Computers, 65(11),* 3304–3317. DOI 10.1109/TC.2016.2532860.

14. Liu, Z., Zhang, Q., Boutaba, R., Liu, Y., Wang, B. (2016). Optima: On-line partitioning skew mitigation for mapreduce with resource adjustment. *Journal of Network and Systems Management, 24(4),* 859–883. DOI 10.1007/s10922-015-9362-8.
15. Liu, Z., Khan, A., Chen, P., Liu, Y., Gong, Z. (2016). Dynamicadjust: Dynamic resource adjustment for mitigating skew in mapreduce. *IEICE Transactions on Information and Systems, 99(6),* 1686–1689. DOI 10.1587/transinf.2015EDL8255.
16. Afrati, F., Stasinopoulos, N., Ullman, J. D., Vassilakopoulos, A. (2018). Sharesskew: An algorithm to handle skew for joins in mapreduce. *Information Systems, 77,* 129–150.
17. Computing, A., Zacheilas, N., Kalogeraki, V. (2014). Real-time scheduling of skewed mapreduce jobs in heterogeneous environments. *Proceedings of 11th International Conference on Autonomic Computing,* pp. 189–200, Philadelphia, USA.
18. Dean, J., Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Communications of the ACM, 51(1),* 107–113. DOI 10.1145/1327452.1327492.
19. Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Seth, S. (2013). Apache hadoop yarn: Yet another resource negotiator. *Proceedings of the 4th ACM Annual Symposium on Cloud Computing,* vol. 1, pp. 1–16.
20. Ibrahim, S., Jin, H., Lu, L., He, B., Antoniu, G. et al. (2013). Handling partitioning skew in mapreduce using LEEN. *Peer-to-Peer Networking and Applications, 6(4),* 409–424. DOI 10.1007/s12083-013-0213-7.
21. Liu, Z., Zhang, Q., Zhani, M. F., Boutaba, R., Liu, Y. et al. (2015). Dreams: Dynamic resource allocation for mapreduce with data skew. *Proceedings of 2015 IFIP/IEEE International Symposium on Integrated Network Management,* pp. 18–26, Ottawa, Canada.
22. Le, Y., Liu, J., Ergun, F., Wang, D. (2014). Online load balancing for mapreduce with skewed data input. *Proceedings of 2014 IEEE INFOCOM,* Toronto, Canada.
23. Ananthanarayanan, G., Kandula, S., Greenberg, A. G., Stoica, I., Lu, Y. et al. (2010). Reining in the outliers in map-reduce clusters using Mantri. *Proceedings of OSDI,* pp. 265–278, Vancouver, Canada.
24. Verma, A., Cherkasova, L., Campbell, R. H. (2011). Aria: Automatic resource inference and allocation for mapreduce environments. *Proceedings of the 8th ACM international conference on Autonomic computing,* pp. 235–244, Karlsruhe, Germany.
25. Zhang, Q., Zhani, M. F., Yang, Y., Boutaba, R., Wong, B. (2015). PRISM: Fine grained resource-aware scheduling for mapreduce. *IEEE Transactions on Cloud Computing, 3(2),* 182–194. DOI 10.1109/TCC.2014.2379096.
26. Jalaparti, V., Ballani, H., Costa, P., Karagiannis, T., Rowstron, A. (2012). Bridging the tenant-provider gap in cloud services. *Proceedings of the 3rd ACM Symposium on Cloud Computing,* vol. 10, pp. 1–14.
27. Apache hadoop yarn. (2020). https://hadoop.apache.org/docs/current.
28. Ahmad, F., Lee, S., Thottethodi, M., Vijaykumar, T. (2012). *Puma: Purdue mapreduce benchmarks suite*. Technical Report, Purdue University, pp. 1–7.
29. Lin, J. (2020). Cloud9: A hadoop toolkit for working with big data. https://lintool.github.io/Cloud9/docs/content/pagerank.html.
30. Hammoud, M., Rehman, M. S., Sakr, M. F. (2012). Center-of-gravity reduce task scheduling to lower mapreduce network traffic. *Proceedings of IEEE 5th International Conference on Cloud Computing,* pp. 49–58, Honolulu, USA.
31. Irandoost, M. A., Rahmani, A. M., Setayeshi, S. (2019). Learning automata based algorithms for mapreduce data skewness handling. *Journal of Supercomputing, 75(10),* 6488–6516. DOI 10.1007/s11227-019-02855-0.
32. Su, S., Tian, Z., Liang, S., Li, S., Du, S. et al. (2020). A reputation management scheme for efficient malicious vehicle identification over 5G networks. *IEEE Wireless Communications, 27(3),* 46–52. DOI 10.1109/MWC.001.1900456.
33. Tian, Z., Gao, X., Su, S., Qiu, J. (2019). Vcash: A novel reputation framework for identifying denial of traffic service in Internet of connected vehicles. *IEEE Internet of Things Journal, 7(5),* 3901–3909. DOI 10.1109/JIOT.2019.2951620.
34. Liu, Z., Wang, X., Shen, L., Zhao, S., Cong, Y. et al. (2019). Mission oriented miniature fixed-wing UAV swarms: A multi-layered and distributed architecture. *IEEE Transactions on Systems, Man, and Cybernetics*, early access. DOI 10.1109/TSMC.2020.3033935.

35. Polo, J., Castillo, C., Carrera, D., Becerra, Y. Whalley, I. et al. (2011). Resource-aware adaptive scheduling for mapreduce clusters. *Proceedings of Middleware,* pp. 187–207, Lisbon, Portugal.
36. Sharma, B., Prabhakar, R., Lim, S., Kandemir, M. T., Das, C. R. (2012). MROrchestrator: A fine-grained resource orchestration framework for mapreduce clusters. *Proceedings of IEEE 5th International Conference on Cloud Computing,* pp. 1–8, Honolulu, USA.
37. Tang, Z., Liu, M., Ammar, A., Li, K., Li, K. (2016). An optimized mapreduce workflow scheduling algorithm for heterogeneous computing. *Journal of Supercomputing, 72(6),* 2059–2079. DOI 10.1007/s11227-014-1335-2.
38. Song, J., He, H., Wang, Z., Yu, G., Pierson, J. M. (2018). Modulo based data placement algorithm for energy consumption optimization of mapreduce system. *Journal of Grid Computing, 16(3),* 409–424. DOI 10.1007/s10723-016-9370-2.
39. Rathinaraja, J., Ananthanarayana, V., Paul, A. (2019). Dynamic ranking-based mapreduce job scheduler to exploit heterogeneous performance in a virtualized environment. *The Journal of Supercomputing, 75(11),* 7520–7549. DOI 10.1007/s11227-019-02960-0.