Cluster Computing (2021) 24:3381–3403
https://doi.org/10.1007/s10586-021-03339-8

A classification of hadoop job schedulers based on performance optimization approaches

Rana Ghazali[1] · Sahar Adabi[1] · Douglas G. Down[2] · Ali Movaghar[3]

Received: 16 January 2021 / Revised: 2 June 2021 / Accepted: 9 June 2021 / Published online: 18 June 2021

Abstract
Job scheduling in MapReduce plays a vital role in Hadoop performance. In recent years, many researchers have presented job scheduler algorithms to improve Hadoop performance. Designing a job scheduler that minimizes job execution time with maximum resource utilization is not a straightforward task. The primary purpose of this paper is to investigate agents affecting job scheduler efficiency and present a novel classification for job schedulers based on these factors. We provide a comprehensive overview of existing job schedulers in each group, evaluating their approaches, their effects on Hadoop performance, and comparing their advantages and disadvantages. Finally, we provide recommendations on choosing a preferred job scheduler in different environments for improving Hadoop performance.

Keywords MapReduce · Job scheduler · Performance metric · Straggler · Data locality · Resource utilization

1 Introduction

Hadoop is an open-source framework for the parallel processing of big data in a distributed environment. Its two main components are HDFS and MapReduce. HDFS, the Hadoop distributed file system, divides input data into blocks of identical size and distributes these blocks between data nodes. MapReduce is a programming model for parallel processing on a cluster of computers in a distributed environment. Jobs consist of two types of tasks: map tasks and reduce tasks. Map tasks convert input data into $<$ key, value $>$ pairs that are referred to as intermediate data. Reduce tasks take these intermediate data as input, then merge values with identical keys to generate final results. A key idea of MapReduce is to transport processes instead of data because large dataset transportation generates excessive network traffic.

Hadoop has a master/slave architecture. Each slave/worker node has a fixed number of map slots and reduce slots in which it can run tasks, and it sends a heartbeat message every few seconds that reports the number of free map and reduce slots to the master node. In order to avoid data transportation through the network, the job scheduler has the responsibility to assign tasks to worker nodes that contain the necessary input data. The primary purposes of the job scheduler are to minimize job completion time and overhead while maximizing throughput and resource utilization. Therefore, the job scheduler has an essential role in Hadoop performance.

FIFO is the default job scheduler in Hadoop. A job is divided into different tasks, and tasks wait in a queue in the order that they arrived. FIFO then selects the first task from the queue and assigns it to the first idle worker node. When processing is complete, FIFO selects the next job in the queue. Its implementation is easy, but it has the following limitations:

- It does not consider job priority, job size, or data locality.

✉ Sahar Adabi
Sahar_adabi@iau-tnb.ac.ir

Rana Ghazali
R_ghazali@iau-tnb.ac.ir

Douglas G. Down
Downd@mcmaster.ca

Ali Movaghar
Movaghar@sharif.edu

[1] Department of Computer Engineering, North Tehran Branch, Islamic Azad University, Tehran, Iran

[2] McMaster University, 1280 Main St W, Hamilton, ON, Canada

[3] Sharif University of Technology, Azadi Ave, Tehran, Iran

- It does not yield a balanced allocation of resources between small and large jobs, so it is possible that starvation occurs for small jobs.

In recent years, many researchers have proposed job schedulers to overcome these limitations, and they have proposed various job scheduler mechanisms that attempt to improve Hadoop performance. Two fundamental metrics that are used to measure Hadoop performance are job execution time and resource utilization. Therefore, the factors affecting these two criteria should be identified, and solutions should be provided to optimize them. The goal is to find job schedulers that minimize job completion time with maximum resource efficiency.

There are several surveys [1–4] of existing job schedulers that discuss their features, advantages, and limitations. They have classified job schedulers based on different aspects: strategy (static/dynamic), environment (homogenous/heterogeneous), time (deadline/delay), etc. [1, 2]. These classifications are not comprehensive and do not consider performance issues, therefore we present a state-of-the-art classification based on performance optimization approaches. In this article, we provide a comprehensive overview of job scheduler strategies and investigate them in terms of performance metrics and their impact on improving Hadoop performance; as a consequence, we propose a novel classification based on the approaches to address performance criteria. Finally, we present guidelines to select a preferred job scheduler in particular settings. Our contributions in this paper are:

- We identify important factors that affect the execution time and resource utilization. Our focus is on straggler tasks, data locality, and resource allocation. We describe the mechanisms that are applied by job schedulers for addressing these factors, investigating their impact on performance.
- A novel classification of job schedulers is proposed, categorized into three main groups: job schedulers for mitigating stragglers**,** job schedulers for improving data locality, job schedulers for improving resource utilization.
- For each job scheduler considered, its approach to improving performance is described and strengths and weaknesses are discussed.
- We investigate the effect of each job scheduler and suggest the preferred one in each group.
- We provide novel guidelines for selecting an appropriate job scheduler for different environmental features.

The rest of this article is organized as follows: "Performance aspects" introduces two performance metrics and their impact on Hadoop applications. We classify existing scheduler mechanisms based on their approach to improving Hadoop performance in "Classification of job scheduler mechanisms based on approaches for improving performance". "Overview of job schedulers for mitigating stragglers" considers mechanisms for preventing stragglers and reducing execution time. In "Overview of job schedulers for improving data locality", we describe and compare job scheduler strategies for improving data locality. "Overview of job schedulers for improving resource utilization" presents and compares job scheduler policies for improved resource usage. We present guidelines to select a preferred job scheduler in different operational environments in "Guidelines for job scheduler selection".

2 Performance aspects

The job scheduler has a vital role in Hadoop performance, and there are various performance aspects. We consider two essential aspects: job execution time and resource utilization, which influence job response time (the time to complete all of the tasks in a job) and resource efficiency. Both of these aspects of Hadoop's performance could be improved by reducing execution time and optimizing the use of resources. To this end, key factors that increase the execution time and result in inefficient use of resources must be identified, as well as mechanisms that are applied by job scheduler policies for addressing these factors. In Hadoop systems, there are three main issues that can lead to increased job execution time and decreased resource efficiency:

1. **Stragglers**: A job ends when all its tasks are completed, but task execution times differ on different nodes due to the various computing capacities of the nodes. Tasks that take longer to execute are known as stragglers. These stragglers lead to prolonging the execution time.
2. **Non-data locality**: If a task is assigned to a node that does not contain the necessary input data, the required data must be transferred from a remote data node, which increases execution time.
3. **Unbalanced resource allocation**: If a large job uses a large share of resources in a cluster, then small jobs may be starved. Therefore, resources should be fairly distributed among the jobs in the cluster.

For each of the issues described above, there are corresponding mechanisms that can be applied by the job scheduler.

1. **Speculative execution**: This mechanism attempts to identify slow nodes and slow tasks in order to avoid the occurrence of a straggler. When stragglers are

identified, their impact is mitigated; for example, a backup task for each slow task may be assigned to fast nodes.

2. **Data locality**: Job scheduler policies should assign tasks to a worker node with the required input data to avoid the increase in execution time that results from data transportation through the network. If this is not possible, the distance that the data must travel should be minimized.
3. **Fair distribution of resources**: The job scheduler mechanism should distribute resources between tasks based on their demand to increase resource utilization and avoid resource wastage.

We will discuss each of these issues in more detail in the following sections.

2.1 Speculative execution and its impact on performance

Nodes in a cluster have different computing capacities and resources; hence the execution time of a task can differ across nodes. In addition, in the MapReduce processing model, the output of the map tasks is the input of the reduce tasks; thus, a reduce task is commenced after finishing all associated map tasks. A task that takes significantly longer to execute than other tasks associated with the same job is known as a straggler. When a straggler occurs, the other tasks must wait, increasing execution time and decreasing cluster throughput; the reduced performance can be unacceptable.

One approach to deal with stragglers is speculative execution, which attempts to identify slow nodes and slow tasks and launches backup tasks on fast nodes to compensate. Two techniques are used for diagnosing stragglers and decreasing their effect on performance: reactive and proactive. In reactive techniques, the job scheduler waits until the straggler is identified then runs a copy of the slow task on a fast node. Proactive techniques attempt to predict the occurrence of stragglers, often with the use of machine learning methods.

2.2 Data locality and its impact on performance

One motivation for MapReduce is to transport processes instead of data. Ideally, the job scheduler should assign tasks to a worker node that contains the requested input data for processing. The concept of data locality involves the distance between the computing node that is processing the task and the data node that contains the corresponding input data. By increasing this distance, data must be transported through more network resources, resulting in high communication costs and increased network traffic; consequently, data transportation time and job execution time are increased. The best case is to assign a task to a worker node that contains the required input data; however, all such nodes are sometimes busy or do not have a free slot. In this case, the scheduler should consider neighboring nodes that have the shortest distance to reduce data transportation delay.

There are three levels of data locality: the first level is node locality, in which tasks are assigned to a worker node that contains the required input data. The second level is rack locality, in which the scheduler assigns tasks to a worker node in a rack that contains the required input data. The last level is off-rack locality, in which the scheduler assigns the task to a machine in the rack that does not contain the required input data.

We can measure data locality with the data locality rate, the ratio of the number of local tasks to the total number of tasks. Different environmental factors influence this measure.

- **The number of tasks**: data locality rate is inversely related to the number of tasks, as increasing the total number of tasks decreases the probability of assigning tasks to a local node.
- **The number of free slots**: The data locality rate and the number of free slots have a direct relationship. By increasing the number of slots, we have more free slots for tasks to be assigned to a node that has the required input data.
- **Data replication factor**: Data replication factor determines the number of copies of the data. By increasing this factor, the probability of finding a node with both the required data and a free slot increases.
- **Cluster size**: By increasing the number of nodes in a cluster, the number of free slots increases, but the probability of assigning tasks to nodes that contain the requested data decreases, so the data locality rate drops.

2.3 Fair distribution of resources and its impact on performance

There are typically two types of jobs, CPU-bound and I/O-bound, which require different resources but run simultaneously in the same cluster. The cluster nodes have different resources, including CPU time, memory, storage space on disk, I/O, computational capacity, and job slots. If a job with a low level of resource requirements is allocated to a node with a higher level of resources, resource wastage results, and resource efficiency is decreased. On the other hand, a job cannot be assigned to a node that lacks the required level of resources. It must wait for the required resources, increasing execution time, and decreasing the utilization of resources.

The job scheduler can aim to assign a task to a node to maximize resource utilization; therefore, it should perform resource management by identification of existing resources in each node and job requirements. For this purpose, job schedulers use two strategies: fair distribution of resources and resource-aware mechanisms.

3 Classification of job scheduler mechanisms based on approaches for improving performance

We propose a classification of job schedulers based on approaches for improving Hadoop performance. Corresponding to the discussion in the previous section, we consider three main groups:

- Job schedulers for mitigating stragglers
- Job schedulers for improving data locality
- Job schedulers for improving resource utilization

The first group can be further divided into two subcategories based on the techniques used for solving the straggler problem. These two subcategories are job schedulers that use reactive techniques and job schedulers that use proactive techniques. The second group aims to reduce data transmission and improve job execution time and can be divided into three subcategories based on the scope of data locality. These are data locality for the map task, data locality for the reduce task, and data locality for the job. The third group aims to improve resource utilization and can be divided into two subcategories, fair resource distribution mechanisms and resource-aware mechanisms.

In the following, we describe existing job schedulers in each group and their impact on Hadoop performance. We also discuss their strengths and weaknesses.

4 Overview of job schedulers for mitigating stragglers

As mentioned earlier, improving execution time has a positive effect on performance; solving the straggler problem addresses this directly. The straggler problem can be broken down into three sub-problems: (1) How to reduce the occurrence of straggler tasks? (2) How to identify straggler tasks? (3) How to mitigate the impact of straggler tasks? There are two approaches to answering these questions, reactive and proactive techniques. We classify schedulers into two categories according to which technique they employ, beginning with job schedulers that use reactive techniques.

4.1 Job schedulers employing reactive techniques

In reactive techniques, the job scheduler waits until a straggler occurs, then launches a copy of the straggler on a fast node, with a goal of reducing overall response time. We describe a number of job schedulers that use this technique.

Longest approximation time to end (LATE) [5] was designed by Zaharia et al. in 2008. This scheduler tries to minimize response time by identifying slow tasks that cause excessive resource consumption on a node. The main idea is to identify straggler tasks, then launch a speculative task on an available node such that the execution time is reduced. For this purpose, LATE first approximates the remaining time to finish for each task by using the task progress rate. It uses a fixed weighted combination of the stages of the map and reduce tasks, considering two stages for the map phase (copy input data and map function) and three stages for the reduce phase (copy intermediate data, sort data, and reduce function). Subsequently, it considers two threshold values: a slow node threshold and a slow task threshold for identifying slow nodes and slow tasks. It then ranks the slow tasks based on their progress rate below the slow task threshold. Finally, LATE launches a backup task for the highest-ranked slow task on an available node.

This method can increase resource utilization and decrease response time. However, it has some drawbacks. Due to its static nature and the use of constant weights for the stages, it is not appropriate for heterogeneous environments. LATE also cannot be used in dynamic environments as it does not consider different types of jobs. For launching speculative map tasks, it does not consider data locality, leading to increased data transmission through the network. In some cases, it is unreliable in identifying stragglers—it may not approximate the time to completion of running tasks well. As a result, it may choose the wrong slow task and launch backup tasks for fast tasks leading to poor performance.

Self-adaptive mapreduce scheduling (SAMR) [6] was presented by Chen et al. in 2010 to address the drawbacks of LATE and improve the accuracy of identifying straggler tasks. It calculates the progress rate of tasks dynamically and adapts to changing environments. This method employs historical information stored in each node to tune the weights for the map and reduce stages, updating these weights after each task execution to compute the task progress rate. The result is greater accuracy than LATE. In SAMR, the slow tasks are divided into two groups: slow map tasks and slow reduce tasks; moreover it can distinguish slow nodes. This allows it to launch backup tasks on fast nodes.

The benefits of SAMR include good detection of stragglers, scalability, reduced execution time, reduced resource usage, and the calculation of task progress in a dynamic manner such that it is compatible with environmental changes and appropriate for heterogeneous environments. This approach faces some challenges, for instance, it does not consider some potentially important features such as different types of jobs and size of the dataset when calculating the weights of the map and reduce stages. It does not consider data locality for launching speculative map tasks. It is also necessary to tune some parameters manually, such as parameters for finding slow tasks and slow nodes and the maximum number of backup tasks to launch.

Ananthanarayanan et al. have deployed Mantri [7] in 2010, which monitors tasks by considering real-time progress reports and, as a result, can detect stragglers early in their lifetime and free up resources. For detecting the cause of a straggler, it considers node characteristics, network characteristics, and the job structure. Mantri includes three parts: (1) restarting stragglers, (2) network-aware placement, (3) avoiding recomputation by protecting outputs of valuable tasks. It kills and restarts a straggler or dispatches a speculative copy when appropriate.

A positive feature of Mantri is the early identification of outliers by integrating static information about job structure and the dynamic progress report. This early action allows resources to be released in a timely manner, resulting in better resource utilization. It improves job completion time. A key drawback of Mantri is that there is no guarantee that a backup task will complete earlier as it needs to kill and restart the speculative task on multiple cluster nodes.

Combination re-execution scheduling technology (CREST) [8] was introduced in 2011. In this strategy, instead of running a single backup task, a combination of backup tasks execute on a group of nodes to decrease job response time, so a min–max optimization problem is solved for minimizing job response time. For this purpose, it uses a graph for modeling a Hadoop cluster and re-executes a combination of speculative map tasks by finding a path from the straggler to the available node; it kills the running tasks on the start node and re-executes the straggler on the end node.

Optimizing running time for speculative map tasks and decreasing the job response time are some of the strong points of CREST. It also considers data locality for re-executing speculative tasks on target nodes. It may require launching more than one task, which could lead to resource wastage. It has a time complexity of $O(|V|^2)$ where V is the number of vertices in the graph, as it uses Dijkstra's algorithm. This method is more complex than LATE and can be difficult to control because it runs multiple tasks rather than just one.

Enhanced self-adaptive mapreduce (ESAMR) [9] was developed in 2012 to address the limitations of SAMR by taking many factors into account that could impact the stage weights. It is similar to SAMR in that it uses historical information; however, ESAMR considers job features such as the size of the dataset and the job type for adjusting the weights of each job's stages. ESAMR uses a clustering method (K-means) to classify historical information. It updates the average weight for each group and calculates job execution time on each node based on the cluster's weight; consequently, it can identify slow tasks and slow nodes, and it avoids running slow tasks on slow nodes.

The merits of this technique are that it can identify stragglers with high accuracy and can reduce error in approximating completion time. As a result, it tends to more accurately recognize slow tasks and slow nodes; therefore, it is appropriate for heterogeneous environments. However, the drawbacks of ESAMR are that it requires tuning some parameters such as the percentage of finishing tasks for map tasks and reduce tasks and the number of clusters for the K-means algorithm. The use of the K-means algorithm requires additional overhead and this technique is limited to this clustering algorithm.

MapReduce reinforcement learning (MRRL) [10] was presented in 2015. This job scheduler uses a classical reinforcement learning algorithm, SARSA, for finding straggler tasks in a heterogeneous environment. It first calculates a progress score and time to complete for both map tasks and reduce tasks. A reward function is used in training a model to allocate slow tasks to fast nodes.

The great advantage of MRRL is that it can produce a model of the system without any prior knowledge about environment characteristics; thus, it is suitable for unknown environments. It can reduce job execution time. This method does take some time to learn the model from interacting with the environment, resulting in significant overhead.

Tolhit [11] was introduced in 2016. It is similar to ESAMR, the main difference being that Tolhit uses a genetic algorithm for classifying historical data on each node. It can recognize slow map tasks and reduce tasks by calculating their progress; a backup task is then launched for a slow task on an optimal node that has high resource efficiency and the least distance to the data node. It uses NwGraph and ResourceInfo to determine optimal nodes for backup tasks. NwGraph is a data structure that contains information about the cluster, such as the number of nodes in the cluster and their minimum distance from the scheduler. ResourceInfo includes resource utilization information for the cluster nodes, such as memory and disk utilization.

By considering data locality for backup tasks and selecting an optimal node for processing these tasks, the job execution time can be reduced. Tolhit uses a genetic algorithm to avoid achieving locally optimal solutions. The disadvantages of this method are that it does not consider some potentially useful features for selecting the optimal node, and it requires some parameter tuning, such as threshold values for the map and reduce task progress.

Progress and feedback based speculative execution algorithm (PFSE) [12], from 2017, is based on LATE. The main idea of PFSE is to improve the estimation of the task execution time in order to reduce the number of unnecessary backup tasks. It uses the current phase and feedback from the completed phases to estimate the completion time of running tasks.

This method has high accuracy in estimating task completion times, reduces unnecessary backup tasks, and avoids resource wastage; however, this strategy focuses only on map tasks, which is a significant limitation.

4.2 Job schedulers with proactive techniques

In proactive techniques, the job scheduler predicts the likelihood of a straggler event. Consequently, it can identify slow tasks and slow nodes before the job scheduler makes its assignments. In this manner, it can avoid the occurrence of stragglers. In what follows, we discuss job schedulers that employ this mechanism.

Dolly [13] was developed by Ananthanarayananet et al. in 2013. It attempts to forecast stragglers, launching multiple clones of every task of a job and using the result of the clone that completes first. Cloning of small jobs can be accomplished with little overhead, and typically the majority of the tasks are small. Delay assignment is a technique for solving the problem of contention for intermediate data between extra clones, by the usage of a cost–benefit analysis.

When the majority of jobs are small, the small jobs consume a small fraction of the resources, resulting in resource efficiency and little overhead. Extending Dolly in a cluster that contains multiple computational frameworks is a challenge, as the workloads tend to have highly variable job sizes.

Yadwadkar et al. presented Wrangler [14] in 2014. It is a system that learns to predict nodes that create stragglers by using an interpretable linear modeling technique and avoids creating stragglers by rejecting bad placement decisions. Wrangler includes two primary components: (1) Model builder: By using job logs and resource usage counters that are regularly collected from the worker nodes, a model is built for each node. These models predict if a task will become a straggler given its execution environment; they also attach a confidence measure to their predictions. (2) Model-informed scheduler: Using the predictions from the models built in the first step a model-informed scheduler then selectively delays the start of task execution if that node is predicted (exceeding the minimum required confidence) to create a straggler. This reduces the likelihood of creating stragglers by preventing nodes from becoming overloaded.

While Wrangler reduces both job completion times and resource consumption, the fact that it builds separate models for each node and workload creates some challenges. In particular, each new node and workload requires new training data, which can be extremely time-consuming.

Speculative execution algorithm based on decision tree (SECDT) [15] was introduced in 2015. With the assistance of decision trees, it considers node attributes and execution information to discover similar nodes and better predict the execution time. CPU speed, memory, the quantity of input data, and network transmission speed are influential factors in task execution. These characteristics with weights are stored and used to construct the tree iteratively. First, the information for each node is mapped to the decision tree then searches in each branch are performed for estimating weights and calculating the remaining execution time.

One problem with this method is that traversing the tree can be time-consuming due to the large amount of stored information. To address this, the algorithm clears stored information at regular intervals. An additional problem is that the pruning of branches causes information loss and, consequently, can lead to incorrect estimation of the remaining time.

Multi-task learning (MTL) [16] was proposed by Yadwadkar et al. in 2016 to reduce training time by sharing information between nodes and workloads. Each task has its own training data set, although typically all training points of all tasks live in the same feature space. The goal of MTL is to leverage this relationship to improve the performance or generalization of all the tasks.

Reducing parameters and training data, improving generalization, facilitating interpretability, and more accurate prediction of stragglers are plus points of this technique. On the other hand, it is embedded in an environment of related tasks, and the learner aims to leverage similarities between the tasks and share this information. The result is significant computational overhead and memory usage.

4.3 Analysis and comparison of job schedulers

In this section, we analyze the schedulers introduced so far and compare them based on their features and limitations.

Job schedulers that use reactive techniques to avoid stragglers include two steps: straggler identification and launching speculative tasks on fast nodes. On the other hand, job schedulers that use proactive techniques to avoid stragglers include one step: straggler identification.

There are two main challenges in reactive techniques; we compare job schedulers based on these. The first challenge is how to determine the weights of different stages for a job to recognize slow tasks. *LATE* uses constant weights in calculating the progress rate for each task; this approach is static and is not suitable for heterogeneous environments. *SAMR* uses historical information, but it does not consider some potentially significant job features such as job type and size of data. *ESAMR* first classifies the historical information from each node by use of K-means classification then applies some features to calculate stage weights, but it is limited to K-means classification. However, this strategy has high reported accuracy for identifying stragglers. *Tolhit* uses historical information and performs classification by the use of a genetic algorithm. *Mantri* monitors the system based on real-time progress reports and estimates the remaining time. In *SECDT*, information from each node is mapped to a decision tree using the constructed branch weights; it then re-executes more than one task on target nodes. *MRRL* uses reinforcement learning to identify straggler tasks and launches the backup tasks on fast nodes. No prior knowledge is required about the environment. *PFSE* uses feedback information received from completed tasks in addition to the progress of currently processing tasks to approximate task completion times and to identify stragglers. The accuracy of the estimation increases as the number of completed tasks increases.

The second challenge in reactive techniques is how to select a target node on which to launch backup tasks. *LATE* and *PFSE* do not consider any features for target nodes. *SAMR*, *ESAMR*, and *MRRL* select a fast node for launching backup tasks by classification of nodes into two groups, slow and fast. *CREST* selects target nodes considering data locality. *Tolhit* chooses an optimal node by use of network characteristics and resource information.

We choose the preferred job scheduler for identifying straggler tasks with high accuracy and reliability, and we select the preferred job scheduler for launching speculative tasks by considering data locality and the determination of fast nodes. *ESAMR* and *PFSE* have the highest accuracy in straggler identification, but the accuracy of *PFSE* increases as the number of completed tasks from the same job increases; therefore, we suggest that *ESAMR* is preferred for straggler identification. *Tolhit* is preferred for selecting target nodes that launch backup tasks because it uses network and resource information to identify an optimal target node.

Reactive techniques rely on a wait-speculative re-execution mechanism; therefore, they lead to delayed straggler detection and inefficient resource utilization. Proactive techniques have a superior effect on performance as they predict stragglers before they happen; therefore, they avoid resource wastage. Recent approaches have used machine learning methods to predict situations that result in stragglers. *Dolly* clones small jobs; on the other hand, the other job schedulers considered use different machine learning methods for predicting stragglers. For example, *SECDT* uses a decision tree that increases in size as the number of nodes in the cluster increases. *Wrangler* predicts stragglers by the use of linear modeling and Support Vector Machines but requires significant training time because it learns a model for each node and workload. *MTL* is similar to *Wrangler*, but it shares data between different models based on the similarity of the tasks; therefore, it significantly reduces the training time, from four hours to 40 min in the scenarios considered in [16].

We choose the preferred job scheduler for predicting straggler tasks with respect to two aspects: high accuracy in prediction and low training time. *Wrangler* and *MTL* have high accuracy and reliability in straggler prediction; however, *Wrangler* has a longer training time than *MTL*. As a result, the preferred existing proactive technique appears to be *MTL*.

4.4 Straggler detection and mitigation: performance impact

As mentioned earlier, straggler tasks can increase execution time, with a resulting negative effect on performance; therefore, many job schedulers are designed to mitigate the performance impact of these tasks. In this section, we consider different performance factors and investigate the impact of the previously discussed schedulers on these factors based on their reported experimental results.

Table 1 presents the evaluation of these schedulers concerning the performance metrics. We briefly explain some of these parameters. Load balancing means that tasks are assigned to all nodes such that all processing nodes are in use as much as possible. Resource sharing reflects whether resources are divided fairly among jobs. Execution time is the time to complete all of the tasks in a job. Execution time is categorized into three levels, according to the improvement over *FIFO*: Low (less than 30%), Medium (between 30 and 60%), and High (greater than 60%). Overhead is any combination of excess or indirect computation time, memory, bandwidth, or other resources that are required to perform a specific task. Note that these aspects are in general not independent from each other. We also emphasize the final column of Table 1 reflects the experimental results reported in the corresponding

Table 1 Comparison of job scheduling algorithms with respect to performance metrics

Job scheduler	Load balancing	Data locality	Resource sharing	Execution time	Overhead	Effect on performance (experimental results)
LATE	Avoids overloading the network	No	By considering a limited number of speculative tasks	Low	No	On average 27% faster than Hadoop's native scheduler (FIFO) [5]
SAMR	Selecting fast nodes for speculative tasks	No	By classifying slow nodes into map and reduce slow nodes	Medium	Yes (historical info.)	Reduces response time by up to 25% compared with Hadoop and up to 14% compared with LATE [6]
Mantri	No	No	By use of resource-aware techniques	Medium	Medium	Improves job completion times by 32% compared with LATE [7]
CREST	Yes	Yes	By launching a group of speculative tasks	High	Yes (graph)	Reduces execution time by 70% (best case) and 50% (average case) compared with LATE [8]
ESAMR	Yes	No	By classifying slow nodes into map and reduce slow nodes	Medium	Yes (K-means)	Decreases the execution time by up to 27% compared with LATE [9]
MRRL	By sufficient exploitation and exploration	No	By considering slow and fast nodes in the reward function	N/A	No	Has not been implemented
Tolhit	By use of resource information	Yes	By use of network information and resource information	Medium	Yes (network and resource info.)	Improvement of approximately 27% in execution time over the Fair scheduler [11]
PFSE	No	No	By reducing unnecessary backup tasks	N/A	No	Reduces unnecessary backup tasks [12]
Dolly	No	No	No	Medium	No	Small jobs sped up by 34–46% compared with LATE [13]
Wrangler	Yes	No	By resource utilization counter	High	Yes (training model for each node)	Improves resource consumption by up to 55% and job completion time by up to 61% compared to speculative execution [14]
SECDT	Yes	No	Yes	Medium	Yes (decision tree)	Improves job execution time [15]
MTL	No	No	By sharing information between related tasks	High	Yes (training data)	Improves job completion time up to 59% and reduces resource usage by up to 40% compared with Wrangler [16]

reference. With this in mind, one must exhibit caution in using them for cross-comparison, but with that caveat, we believe that the given information is instructive.

As can be seen in Table 1, reactive methods improve execution time by 30% on average; however, proactive techniques improve execution time by 50% on average. *CREST* has the greatest impact on performance in the reactive technique category, reducing execution time by up to 50% on average because it runs a group of speculative tasks instead of one task; however, it has higher complexity. *Wrangler* has the best impact on performance in the proactive techniques group and decreases completion time by up to 61%; however, it requires significant training time.

5 Overview of job schedulers for improving data locality

As mentioned earlier, improved data locality has a positive effect on performance because it avoids data transportation through the network, resulting in decreased execution time. Many researchers have proposed job schedulers for improving data locality. Here we categorize such schedulers into three classes based on the type of task for which data locality is considered: data locality for map tasks, data locality for reduce tasks, and data locality for jobs (both map tasks and reduce tasks). In the following, we describe the job scheduler strategies in each group and discuss their advantages and disadvantages.

5.1 Job schedulers with data locality for map tasks

In these approaches, the job scheduler attempts to increase the number of local map tasks, those that are assigned to nodes with the required data present.

Delay [17] was proposed by Zaharia et al. in 2010. This mechanism uses job postponement to increase the frequency of local tasks. When a node requests a job, if the head-of-line job in the queue cannot start a local task, it should wait, so the job scheduler skips this job and examines subsequent jobs in the queue in order to launch a local task. This strategy considers a threshold value on waiting time to prevent starvation. If the waiting time for a job reaches this threshold value, it allows a non-local task to start.

Its implementation is simple, has little overhead, and avoids starvation. Delay faces some problems, for instance, the threshold value needs to be manually tuned. If its value is too large, then jobs may wait for a long time, so there is a risk of job starvation. On the other hand, if the value is too small, then the number of non-local tasks is increased. This method is not appropriate for jobs that have high execution times and for environments with limited free slots in nodes as this combination will mean it is highly likely that the threshold is exceeded, resulting in poor performance.

Matchmaking [18] was presented by He et al. in 2011 to address some of the limitations of the Delay scheduler. In this technique, nodes use locality markers to ensure that each node has an equal chance to acquire a local map task. In the second heartbeat of the node, if the node could not launch a local map task, then it assigns a non-local task to avoid wasting computing resources. All locality markers will be cleared when a new job is added to the job queue.

The advantages of this method are that it leads to higher data locality rates and better response times for map tasks, and there is no need to tune any parameters. It shares resources fairly. However, this method does not consider rack locality and job priority and does not scale well for large clusters.

Zhang et al. have designed next-K-node scheduling (NKS) [19] for homogeneous environments in 2011. It considers three levels of data locality (node locality, rack locality, off-rack locality) for map tasks. First, it predicts the next K nodes that will request a job by calculating the progress of tasks. Then it estimates the likelihood of data locality for map tasks in these K nodes; finally, it assigns tasks according to these probabilities.

The main benefit of NKS is decreased network load. A key drawback is the need to tune K. If its value is too large, high computational overhead results. If the value is too small, then it may not recognize the correct nodes for task requests. In addition, this mechanism is not appropriate for heterogeneous environments because it does not consider different node features.

Ibrahim et al. introduced Maestro [20] in 2012 for improving data locality for map tasks. The main idea of this method is running map tasks to be aware of data replication; for this purpose, it keeps track of the locations of chunks along with the locations of their replicas, as well as the number of additional chunks hosted by each node. Map task scheduling is done in two steps. In the first step, free slots for each node are filled based on the number of map tasks and the replication scheme of their input data. In the second stage, the probability of scheduling map tasks on a node that contains replicas of its input data is estimated.

Maestro can schedule map tasks with low impact on other nodes, it provides a good distribution of intermediate data in the shuffling phase, and it reduces the number of non-local map tasks. However, this method is only suitable for known environments as we should know how data replication is performed. Maestro has some overhead because it stores information about the location of input data and their replicas.

Bu et al. developed the interference and locality-aware (ILA) [21] scheduling strategy for virtual MapReduce clusters in 2013. The main idea of this strategy is to reduce interference between virtual machines and improve data locality for map tasks by combining interference-aware scheduling and locality-aware scheduling. It consists of four modules: (1) the interference-aware scheduling module (IASM): By using a task performance prediction model, it decreases the interference between running tasks on VMs. (2) the locality-aware scheduling module (LASM) improves data locality for map tasks by using the Adaptive Delay Scheduling algorithm; (3) the task profiler stores the demand of each task and passes task information to the IASM and LASM modules; (4) the ILA scheduler coordinates the IASM and LASM modules to achieve interference-free high-locality task management and collects runtime information from the virtual machines by monitoring their resources.

The strong point of this technique is that it improves data locality for map tasks, and it has high system throughput. The drawback of this strategy is that it does not consider HDFS interference and has significant overhead.

High-performance scheduling optimizer (HPSO) [22] was deployed by Sun et al. in 2015. This strategy attempts to improve data locality for map tasks by the use of a prefetching mechanism. Suitable nodes for future map tasks are predicted depending on the current pending tasks and the required data is preloaded to memory to avoid delay in launching new tasks. First, it forecasts free and occupied slots by the use of predictive modules that compute the remaining time of map tasks, but it does not use historical information (as is done for cloud computing environments). It then prefetches necessary input data for map tasks from a remote data node or local disk to avoid delay in starting new tasks.

This strategy reduces network overhead by overlapping data transmission for the next map task with data processing of the running map task and reduces waiting times of map tasks with rack locality and off-rack locality. The main challenge of the prefetching mechanism is determining which data should be prefetched. The prefetching accuracy is an essential factor that limits its performance.

Wang et al. designed the Joint [23] scheduler in 2016. Joint is a scheduler /routing algorithm that views the data locality problem from a network perspective and attempts to solve it using routing data from the cluster's communication network. It uses a queueing architecture that captures both the data transmission in the communication network and the task execution by machines, and it uses the shortest queue policy to assign incoming tasks to nodes and to route tasks in the communication network.

This strategy improves the throughput and performance because it utilizes resources efficiently by balancing the tasks assigned to local machines and remote nodes. It avoids network congestion by balancing the traffic load. This method is not applicable in more heterogeneous environments, as it does not consider different link bandwidths and the various structures of the connections between machines.

In 2017, Benifa et al. proposed efficient locality and replica aware scheduling (ELRSA) [24] to improve data locality and to perform consistently in heterogeneous environments. It consists of two parts: (1) A data placement strategy is formulated to compute dynamically the available space in the nodes for positioning the data. The task scheduling algorithm is designed to satisfy data locality constraints and to place the task in a node that holds the data. (2) An autonomous replication scheme (ARS), which decides data objects that should be replicated by considering their popularity and replicas that should be removed when they are idle.

ELRAS improves throughput, data locality, and reduces execution time and cross-rack communication. This algorithm is simple and adapts to dynamic environments if new nodes are added or removed. However, the computational overhead is significant.

In 2018, Merabet et al. introduced the predictive map task scheduler [25] for optimizing data locality for map tasks. It uses a linear regression model for predicting execution times of map tasks and future free slots on all nodes in the cluster. It consists of two main modules: (1) A task scheduler that uses the information sent by predictive and prefetching agents to build a scheduling scheme for future map tasks for high data locality; (2) A prefetching manager that collects information from prefetching agents about input data present in all nodes.

This method improves data locality and execution time for map tasks; however, it is more useful for large jobs because by increasing the number of map tasks, prediction accuracy is increased. The drawback of this method is that it needs training data, and it has low accuracy in predicting execution times when job sizes are small.

Hybrid scheduling mapreduce priority (HybSMRP) [26] was presented by Ghandomi et al. in 2019 and is a hybrid scheduler that combines dynamic job priority and data localization. It determines job priority based on three parameters: running time, job size, and waiting time. HybSMRP uses a localization marker for each node to give a fair chance to be assigned a local task. After two unsuccessful attempts to obtain a local task, a node obtains a non-local task.

Increasing the data locality rate for map tasks, decreasing completion time, and avoiding wastage of resources are merits of this approach. However, it does not consider some environmental features for job priority and it assumes only one queue for jobs.

5.2 Job schedulers with data locality for reduce tasks

In the MapReduce framework, we are aware of the location of input data for map tasks, and we can use this information to schedule map tasks. In contrast, the location of intermediate data that is generated by map tasks is unknown. Therefore, data locality for reduce tasks is a challenge. Despite this challenge, there are several job schedulers that consider data locality for reduce tasks. In this section, we discuss job scheduler strategies that use this approach.

Locality-aware reduce task scheduler (LARTS) [27] was proposed by Hammoud et al. in 2011. It is a strategy that is aware of the size and network partition locations. This mechanism uses early shuffling to improve performance, so it starts scheduling reduce tasks after completion of a particular proportion of map tasks. This strategy suggests scheduling reduce tasks on a maximum-node in a maximum-rack. A maximum-rack of reduce task R is a rack that maintains one or more data partitions for R with an aggregate size larger than the partitions held at other racks. A node that holds the largest partition for R at the maximum-rack is defined as a maximum-node.

This method has some positive points, for example, it improves the three levels of data locality (node locality, rack locality, and off-rack locality) for reduce tasks and decreases network traffic. It has some weak points such as reduced system utilization and a low degree of parallelism.

Hammoud et al. have developed the center-of-gravity reduce scheduler (CoGRS) [28] in 2012. The main idea of this strategy is to reduce network traffic by assigning reduce tasks to the center of gravity node determined by network location. It combines data locality awareness, and partition skew awareness for scheduling reduce tasks.

This method can reduce execution time and off-rack network traffic, in particular when the partition skew is high, but it does not consider network congestion and slot utilization for simultaneous jobs.

In 2014, Arslan et al. designed LONARS [29] to improve data locality of reduce tasks and reduce network traffic for data-intensive applications. It considers locality and network-awareness for scheduling reduce tasks. First, it defines a cost function to determine the cost of assigning reduce tasks to each node; it then uses the K-means algorithm to group nodes based on the cost function. Finally, it schedules reduce tasks to a node with minimum cost.

The advantages of this technique are optimizing the shuffle phase and reducing network traffic. The disadvantage of this method is that it introduces some overhead as it requires network information such as bandwidth capacity.

5.3 Job schedulers with data locality for jobs

In these approaches, data locality is considered for both map and reduce tasks. In this part, we discuss several job schedulers that use this approach.

In 2013, Wang et al. have proposed a scheduling mechanism to improve the data locality of tasks by the use of online simulations and predictive mechanisms [30]. This mechanism consists of two modules: (1) A task predictor that uses system state information to predict future events such as data node availability and job completion time; (2) A job simulator that simulates the behavior of the scheduler. This mechanism has two limitations. First, it uses a simple linear regression model that only considers input data size. As a result, its estimates tend to be somewhat inaccurate. Second, it only predicts execution time for submitted jobs.

Suresh et al. developed optimal task selection [31] in 2014 that tries to assign tasks with high data locality. It considers several criteria to choose the optimal task for assignment, such as the replica count of the input data, the predicted arrival time of the next task to a node that has free slots, and the load of the disk that contains the input data. This method is not efficient for small jobs, and it does not consider some factors to determine optimal nodes, such as the size of the cluster.

In 2014, multithreading locality [32] was presented to address the data locality problem using a parallel search under a multithreaded architecture. In this method, a cluster is divided into N blocks, and each block is scheduled with a specific thread. When a job arrives, all threads start to search for a node with high data locality. Whenever a thread finds a node, it then informs the other threads to finish their search. If no thread can find an appropriate node, then a non-local task is started. The advantages of this method are high scalability and parallel search, but it does not consider the load of the cluster.

Hybrid job-driven scheduling scheme (JOSS) [33], introduced in 2016, considers data locality for both map tasks and reduce tasks. JOSS classifies MapReduce jobs based on job size and job type into three groups: small job, map-heavy, and reduce-heavy jobs, then designs corresponding scheduling policies. The strong points of JOSS are increased data locality, improved job performance, and avoidance of job starvation. However, JOSS does not consider dynamic, heterogeneous environments.

Deadline guarantee and influence-aware scheduler (DGIA) [34] was introduced in 2018 and includes two stages: (1) A preliminary stage to determine a data locality allocation plan to satisfy the deadline requirements of new tasks. (2) A modification stage to reallocate tasks that cannot meet their deadline requirements. For this purpose, it uses a graph to model network flow and uses a minimum cost maximum flow (MCMF) solution.

The benefits of this strategy are increased data locality and improved resource usage, however there is additional overhead, including the computation time for allocation decisions.

5.4 Analysis and comparison

In this section, we analyze these schedulers and compare them based on their strategy and limitations.

Many researchers focus on data locality for map tasks because there is information about the location of input data. Data locality for reduce tasks is challenging because the output of the map tasks serves as input data for the associated reduce tasks. As a result, there is no prior information about the location of input data for the reduce tasks.

In the first subcategory, *NKS* and *HPSO* cover three levels of data locality (node locality, rack locality, and off-rack locality). There are different strategies for considering data locality in assigning map tasks such as applying delay, replication-aware techniques, prediction, and prefetching.

Each of the mechanisms has some challenges, for example in *Delay*, the delay time needs to be tuned. Data locality-aware and replication-aware mechanisms require information about input data and their replica locations, which generates some overhead. Predictive mechanisms use different machine learning methods to learn how to assign the task to a node considering data locality; therefore, they need training data and require time to learn the model. In the prefetch mechanism, it should be determined which data and when to fetch.

For improving data locality for reduce tasks, a scheduler should be aware of the network location of partitions. It should use this information to find a node that contains input data for reduce tasks, then assign reduce tasks to this node. In this subcategory, all of the methods cover three levels of data locality, and only *LONARS* uses a machine learning method (the K-means algorithm) for clustering nodes. In the third group, prediction, classification, and parallel search are used for improving data locality for both map and reduce tasks. In the prediction method, job information is used to predict job completion time, the next node with a free slot, and the arrival time for the next job, resulting in some overhead. In the classification method, jobs are classified into three groups: small map, heavy map, heavy reduce. These groups are then scheduled based on their characteristics, however, the overall efficiency depends on classification accuracy.

5.5 Impact on hadoop performance

As mentioned earlier, increased data locality has a positive effect on performance. In Table 2, we evaluate these job scheduling algorithms with respect to the performance metrics. The evaluations are based on the reported results of their experiments. The data locality rate has a direct effect on execution time improvement. For job schedulers that consider data locality for map tasks, *Maestro*, *HPSO*, and *ELRSA* have the highest reported data locality rates for their scenarios, more than 80%. *LARTS* has the highest data locality rate among the job schedulers that consider data locality for reduce tasks. *Predictive* is the best strategy amongst those that consider data locality for map tasks, with a reported decrease in execution time of up to 57%. The best strategy in terms of impact on execution time among the job schedulers with data locality in reduce tasks is *CoGRS,* with a reported reduction of the execution time of up to 23.8%. Again, as these numbers are based on the reported experiments, one has to use some caution in making cross-comparisons.

6 Overview of job schedulers for improving resource utilization

As mentioned before, resource utilization means the optimal use of available resources on each node. Some worker nodes do not have enough capacity to perform an assigned task; therefore, such a node cannot continue to execute tasks until the system releases resources, which leads to poor performance. In Hadoop, the resource allocation problem is typically an NP-Complete problem. For this problem, two approaches are introduced: fair resource distribution mechanisms and resource-aware mechanisms.

In the following, we describe job scheduler methods for each approach, their impact on Hadoop performance, and their strengths and weaknesses.

6.1 Job schedulers with fair resource distribution

Some jobs need computational resources, whereas other jobs require I/O resources. Some tasks cannot be completed until resources used to execute other tasks are released, leading to increased response times and poor performance. In this section, we provide an overview of strategies that address this performance issue.

Fair scheduler [35] was introduced in 2011 by Facebook. This scheduler is a method for assigning resources to jobs such that all jobs have equal resource shares. For this purpose, jobs are grouped into pools based on their priority; each user has its own pool with associated minimum resource share. The number of concurrently running jobs per pool and user can be limited. If one pool has idle resources they are divided among other pools to avoid resource wastage and prevent starvation.

This strategy works well in both small and large clusters. By considering job priorities and the fair allocation of resources between jobs, it avoids resource wastage and starvation. However, it does not consider the size of a job, potentially leading to unbalanced performance in the nodes. Pools have a limitation on the number of concurrent jobs.

Capacity scheduler [36] was proposed in 2013 by Yahoo!. This scheduler aims to maximize resource efficiency and cluster throughput. This scheduler is similar to the Fair scheduler, but it uses multiple queues instead of pools and assigns jobs into queues. Resources are divided between these queues. A minimum capacity of resources is guaranteed by limiting the running tasks and jobs from a single queue and scheduling jobs based on their priority. For resource efficiency, resources can be moved from an empty queue to a queue with a heavy load, and after finishing a job, these resources are returned to the main queue.

Some advantages of this scheduler are fair resource allocation, high resource efficiency, improved cluster throughput, and the consideration of job priority. At the same time, it can be difficult to configure, and it can be difficult to select the appropriate queue for job assignments.

Cross-task coordination mapreduce (COOMR) [37] was designed to increase the coordination between tasks to improve the use of shared resources on computation nodes. This scheduler selects consistent jobs that have minimal interference, as task interference can cause prolonged execution time for map tasks. At the same time, excessive

Table 2 Comparison of scheduling algorithms for improving data locality with respect to performance metrics

Job scheduler	Load balancing	Data locality	Resource sharing	Exec. time	Overhead	Effect on performance
Delay	Yes	Map task	No	Medium	No	Improves execution time by 40% relative to FIFO [17]
Matchmaking	No	Map task	Yes	Medium	No	Improves response time by 44% and increases data locality rate compared with FIFO [18]
NKS	Net. load-balance	Map task	No	N/A	Uses task progress	Reduces non-local map task up to 78% and reduces network load up to 77% compared with FIFO [19]
Maestro	Balance intermediate data for the shuffle phase	Map task	Based on hosted map tasks	Medium	Locations of input data and replicas	Improves execution time by up to 34% and improves local map tasks by up to 95% compared with FIFO [20]
ILA	Prevents interference between virtual machines	Map task	Interference-aware scheduling	High	Info. on interference and locality	Speedup by a factor of 1.5–6.5 for individual jobs, improves system throughput by a factor of up to 1.9 and improves data locality by up to 65% compared with Delay [21]
HPSO	Overlaps transmission and processing	Map task	No	N/A	No	Improves data locality by at least 88.7% and improves performance by 6% [22]
Joint	Balances the traffic	Map task	Yes	N/A	Routing info	Improves throughput
ELRSA	No	Map task	No	N/A	Statistics table	Locality rate of approximately 82% [24]
Predictive	Overlaps the task execution and data transfer phases	Map task	No	Medium	Uses linear regression to predict execution time	Improves data locality by at least 73.33% and total execution time by 57% compared with FIFO [25]
HybSMRP	No	Map task	Yes	Low	No	Improves completion time by 11.51% compared to Fair and 29.15% compared to FIFO and has a data locality rate of 58.7% [26]
LARTS	Early shuffling	Reduce task	Early shuffling	N/A	No	Improves localities of node, rack, and off-rack by 34.45, 0.32 and 7.5%, respectively compared with FIFO [27]
CoGRS	No	Reduce task	Yes	Low	Partition Info	Improves execution time up to 23.8%, increases node-local data by 34.5%, and decreases rack locality and off-rack locality by 5.9% and 9.6%, on average compared to FIFO [28]
LONARS	No	Reduce task	No	Low	Network bandwidth info	Increases the rack level shuffle ratio by 22% compared to FIFO, 19% compared to RackAware, and 17% compared to CoGRS. 3–4% improvement in total job completion time compared to FIFO [29]
Online Simulation	Yes	Job	No	N/A	Yes	Has 95% accuracy in prediction [30]
Multi Threading Locality	No	Job	Parallel processing	Medium	No	Improves energy consumption by about 29% compared to Matchmaking on average, about 31% compared to Delay, and about 47% compared to FIFO [32]
JOSS	Yes	Job	No	N/A	No	Improves data locality rate by 33.44, 32.16% compared with FIFO and Fair [33]
DGIA	Yes	Job	Yes	Low	Modeling graph	For large tasks, the number of deadline misses can be reduced by 20–30% [34]

I/O requirements can degrade the disk I/O bandwidth. COOMR consists of three components: (1) Cross task opportunistic memory sharing (COMS) increases the coordination among map tasks with the usage of shared memory, (2) Log-structured I/O consolidation (LOCS) reorganizes intermediate data in the shuffling phase, and (3) A novel merging method for pairs without movement, key-based in-situ merge, reduces the I/O load.

The strengths of COOMR are reduced task interference, improved I/O performance, scalability, and decreased execution time. There is less improvement for map tasks as they have less interference in reading input data from HDFS, the focus is on merging intermediate data for reduce tasks.

DynMR [38] was presented by Tan et al. in 2014. This mechanism uses interleaving for running map and reduce tasks. It consists of three components: (1) Identifying unused resources in the shuffling phase and allocating these resources to the next task; (2) Assembling the reduce tasks in a progressive queue and executing them in an interleaved order; (3) Merging the threads of all partial reduce tasks.

While this strategy improves resource performance and decreases execution time, it focuses only on reduce tasks.

6.2 Job scheduler mechanisms for resource-awareness

These mechanisms are aware of existing resources in each node and the job resource demands; they prevent assigning jobs to nodes with excess resources. Therefore, they avoid resource wastage and reduce task waiting times for obtaining required resources, leading to decreased completion times and improved performance. In this section, we provide an overview of these strategies.

Resource-aware scheduling (RAS) [39] was designed in 2009 and monitors available resources in each node to perform dynamic resource allocation. It consists of three steps: (1) Dynamic free slot advertisement: it calculates available computation slots dynamically in each node based on the associated metrics obtained from monitoring the node, as opposed to considering a fixed number of available slots. (2) Free slot priorities: nodes are sorted such that those with higher resource availability are presented to the job scheduler. (3) Energy-efficient scheduling: the energy consumption in the scheduler is less than for the Hadoop basic scheduler.

This strategy improves resource utilization, and load balancing through dynamic resource allocation leads to significantly reduced job response times. The drawbacks of this mechanism are the lack of support for preemption of reduce tasks, the need for additional monitoring, and the need for forecasting capabilities to manage network bottlenecks. These last two issues create significant overhead.

Resource-aware adaptive scheduler (RAAS) [40] was introduced in 2011 and uses job profiling information to dynamically allocate resources at runtime. This mechanism utilizes the job slot instead of the task slot that is bound to a particular task of the job. This method aims to determine the best placement of tasks to nodes for maximizing resource utilization while considering the completion time goal for each job. It consists of five components: (1) placement algorithm; (2) job utility calculator; (3) task scheduler; (4) job-status updater; (5) completion time estimator.

This technique allocates resources dynamically at runtime and assigns slots for jobs to maximize resource utilization with a consequent improvement in completion times. It relies on the job profile based on information from previous execution, therefore the profile accuracy plays an important role in the performance.

Context-aware scheduler (CASH) [41] was proposed in 2012. In this strategy, jobs are classified into two groups, CPU-bound and I/O-bound, based on information from summary logs. It classifies nodes into two groups, computational or I/O-efficient. It attempts to assign CPU-bound jobs to computational-efficient nodes and I/O-bound jobs to I/O-efficient nodes.

This strategy avoids resource wastage and increases resource efficiency. However, this strategy requires each job to be executed one time to determine its type.

Rasooli and down designed classification and optimization-based scheduler for heterogeneous hadoop systems (COSHH) [42] to achieve competitive performance with minimum resource share satisfaction. for this purpose, it considers heterogeneity in the system and makes scheduling decisions based on fairness and minimum resource shares. COSHH consists of two main components: (1) A queuing process that assigns arriving jobs to the appropriate queue, (2) A routing process, finding the best assignment between a job and free resources. COSHH uses K-means to classify jobs into two classes: I/O-bound and CPU-bound, and it solves a linear programming problem for scheduling decisions.

COSHH has benefits, for instance, minimizing job completion time, high data locality, scalability, and fairness in resource distribution. However, it suffers from the requirement that a significant number of parameters must be estimated, classification is by K-means, and solving a linear program generates some overhead, which increases as a function of the number of resources.

Phase and resource information-aware scheduler for mapreduce (PRISM) [43] was presented for resource-aware scheduling at the phase level of tasks. For this purpose, it divides tasks into multiple phases and uses a phase-based scheduler and a job progress monitor for gathering phase progress information. Its accuracy depends on the profile that contains the resources required for each phase.

The advantages of this strategy are improved resource utilization, decreased completion time, a higher degree of parallelism, and high flexibility. Splitting tasks into an increased number of phases increases the complexity and generates scheduling overhead.

Workload characteristics and resource aware (WCRA) [44], developed in 2015, considers workload status and node capabilities according to a set of performance parameters. Time for data parsing, map operation, sorting, and merging results are considered for map tasks, and for reduce tasks, time to merge, parse, and perform the reduce operation are considered to classify jobs into two categories, CPU-bound, and I/O-bound. For acquiring node capabilities, several node features are considered, such as CPU capacity, I/O performance, and physical memory available, then nodes are categorized into two pools of resources: CPU busy and I/O busy nodes. When a job arrives job demands are obtained by running sample tasks of the job on a small data set, and a job profile is constructed that is used for scheduler decisions. WCRA assigns CPU-bound jobs to I/O busy nodes and vice versa.

The benefits of this method are improved resource utilization, workload balancing, consideration of job priorities, all with a consequence of reduced completion times. The drawback of this method is the need to run sample tasks of a job on a small dataset, generating overhead.

Job allocation scheduler (JAS) [45] was introduced by Hsieh et al. in 2016 for balancing resource utilization. It categorizes jobs into two groups: CPU-bound and I/O-bound by calculating map data throughput as a function of the map input data, map output data, shuffle output data, and shuffle input data. If the throughput is less than the disk average I/O rate, then the job is classified as CPU-bound. The CPU and I/O capabilities of nodes are calculated from the execution times of completed tasks. Jobs are then assigned according to their classification and the node capabilities.

On the positive side, it reduces execution time and balances resource utilization. However, JAS has some potential problems, for instance, it does not consider data locality in assigning tasks, increases the network traffic, and generates some computational overhead.

dynamic grouping integrated neighboring search (DGNS) [46] was developed in 2017 and considers the MapReduce and HDFS layers and heterogeneous workloads and environments. It consists of four phases: (1) Job classification: It divides jobs into two groups, CPU-bound, and I/O-bound. (2) Ratio table creation: It creates a capability ratio table for nodes that contain CPU and I/O slots for each node in the map layer and a capability ratio table for each data node in the HDFS layer. (3) Grouping and data block allocation: It uses a strategy that groups nodes according to CPU-slot numbers and allocates data blocks based on the storage capacity of data nodes. (4) Neighborhood search: it assigns tasks to nodes through a neighborhood search; if the required data block is not present in the first group, it considers non-local nodes.

The main advantages are that it improves performance and has high data locality. However, it does not consider different types of jobs.

Energy-efficient mapreduce scheduling algorithm for YARN (EMRSAY) [47] was proposed in 2020 to minimize energy consumption by considering deadline constraints. This strategy consists of two phases. In the first phase, separate deadlines for map and reduce tasks are estimated. In the second phase, a heuristic is used to calculate the average power consumption for each node in the cluster. Map and reduce tasks are assigned according to the calculated average power consumption for nodes (higher priority to lower values).

Minimizing energy consumption is the main benefit of this method, however, one drawback is related to the dynamic nature of the average power consumption, requiring recalculation at the start of each round, leading to overhead.

6.3 Analysis and comparison of job schedulers

In this section, we analyze these schedulers and compare them based on their strategy and limitations.

As mentioned before, there are two techniques for resource utilization: fair resource distribution mechanisms and resource-aware mechanisms. Fair resource distribution attempts to give equal chances to users or jobs in resource distribution; however, resource-aware mechanisms allocate resources based on job requirements and resource characteristics. Schedulers in the first group do not use information about the characteristics of jobs and available resources; therefore, they have much less overhead. The second group classifies jobs into two groups: I/O-bound and CPU-bound with the use of different methods and based on the capabilities of the resources, an appropriate matching is performed.

In the first group, the two schedulers *Fair* and *Capacity* are extremely popular. These methods are similar, but the *Fair* scheduler uses pools, and fair resource sharing among users, while the *Capacity* scheduler uses multiple queues and fair resource sharing among jobs.

Resource-aware job schedulers consist of three main phases: (1) Gathering information about job demands and resource capabilities; (2) Classification of jobs and nodes based on collected data; (3) Matching jobs and resources. In the first phase, different information is used to obtain job states, for example, *RAS* monitors nodes, *RAAS* and *WCRA* use job profilers, while *DGNS* constructs a ratio table. In the second phase, job classification is done with different methods such as K-Means for *COSHH* and the calculation of job throughput for *JAS*.

6.4 Impact on hadoop performance

As mentioned earlier, increasing resource efficiency and reducing resource wastage have a positive effect on performance. In Table 3, we evaluate job scheduling algorithms that have these concerns with respect to the performance metrics. As before, these observations are based on their reported experimental results.

In the first group, fair resource sharing among jobs reduces the waiting time for resources. *Capacity* has the greatest effect on completion time.

In the second group, resource-aware schedulers, where some information about node capabilities and job demands are known and used for assigning tasks to suitable nodes, a positive effect on performance is observed. *COSHH* and *WCRA* have the best impact on performance however both of them generate significant overhead. Further evaluation of tradeoffs for these two policies is warranted, in particular over a wider range of systems.

7 Guidelines for job scheduler selection

In this section, we consider different environmental features to select a preferred job scheduler for different operating environments. We present a guideline for each classification that determines which job scheduler is suitable for each environment. It is worth mentioning that there are three levels of heterogeneity: cluster, users, and workload. In this section, we consider cluster heterogeneity which means the nodes have different capabilities such as CPU capacity, memory, and bandwidth. Moreover, we consider some features of the job scheduler like flexibility, scalability, and complexity. Depending on the size of the cluster and the workload, we determine three levels for scalability such that low scalability corresponds to both small job and cluster size, medium scalability corresponds to either the job size or cluster size is large (but not both) and high scalability corresponds to job and cluster size both being large. For this purpose, we consider a cluster as a small cluster for less than 50 nodes otherwise it is a large cluster, and input data are defined as small size whenever

Table 3 Comparison of job scheduling algorithms for improving resource utilization with respect to performance metrics

Job scheduler	Load balancing	Data locality	Resource sharing	Overhead	Execution time	Effect on performance (experimental results)
Fair	Yes	No	Yes	No	Medium	Achieves a 37.72% decrease in average completion time compared to FIFO [35]
Capacity	Yes	No	Yes	No	Medium	Improves average completion time by 40% compared to FIFO [36]
COOMR	Yes	No	Yes	No	Medium	Improves performance up to 44%, execution time for reduce tasks as much as 57.4%, and the total execution time by 38.7% compared to FIFO [37]
DynMR	Yes	No	Yes	No	N/A	Improves resource utilization
RAS	Yes	No	Yes	No	N/A	Improves response time and resource utilization
RAAS	Yes	No	Yes	Monitoring	High	Improves completion time by a factor of 1.2–2.5 and improves resource utilization (compared to Fair) [40]
CASH	Yes	No	Yes	Log info	N/A	Improves performance by around 40% and avoids overloading nodes (compared with FIFO) [41]
COSHH	Yes	Yes	Yes	K-means, linear program	High	Improves average completion time by 31.27 and 42.41% over FIFO and Fair [42]
PRISM	Yes	No	Yes	Splitting tasks into phases	High	Improves resource utilization by up to 18% and completion times by up to 1.3 times (compared to Fair [43])
WCRA	Yes	Yes	Yes	Run sample task	Low	30% improvement in performance compared to FIFO, Fair, and Capacity [44]
JAS	No	No	Yes	Computational	Low	Improves performance by nearly 15–18% compared with FIFO [45]
DGNS	Yes	Yes	Yes	Ratio table	Medium	Improves completion time and resource utilization
EMRSAY	Yes	No	Yes	Calculating energy consumption	Medium	Improves energy efficiency by 35% against Delay [47]

its size less than 20 GB else it has a large size. In addition, we determine the complexity according to the computational and job scheduler method's complexity. For convenience, we have categorized the types of environments into four groups: homogenous, heterogeneous with small size jobs, heterogenous with medium size jobs, and heterogeneous with large size jobs. We have then allocated the appropriate job scheduler to each environment category and have selected the preferred one in each group.

7.1 Job scheduler selection for mitigating stragglers

We present a guideline to select the appropriate job scheduler to overcome straggler problems in each environment. Table 4 shows some environmental features and whether particular job schedulers consider them. We have distributed job schedulers among four environmental categories as follows:

1. **Homogenous environment**: *LATE* is the job scheduler that only works well in this environment, due to its static manner in tuning stage weights for computing the remaining time of tasks. It is suitable for jobs where the reducer does more work, and map tasks are a smaller fraction of the total load. Therefore, *LATE* is the preferred choice for this environment because of its simplicity.
2. **Heterogeneous with small size jobs**: *SAMR*, *Mantri*, *CREST*, *ESAMR*, and *Tolhit* from the reactive subcategory and *Dolly* from the proactive subcategory are in this group. *SAMR* is suited to jobs where reduce tasks have longer execution times, while *ESAMR* is suitable for different job types. *CREST* is suitable for small jobs; however, it is not scalable as by increasing the number of nodes in the cluster, the resultant graph will significantly increase the time to find the path, and it cannot launch more than seven tasks. *Mantri* is suitable for a known environment as it requires machine and network characteristics. *Tolhit* is ideal for heterogeneous environments that have information available for the network and resources to launch backup tasks on optimal nodes. *Dolly* is suitable for interactive small jobs; however, it results in resource wastage. Therefore, *ESAMR* appears to be the preferred choice for straggler detection in this environment because of its high accuracy, and *Dolly* is preferred for straggler prediction.
3. **Heterogeneous with medium size jobs**: *MRRL* from the reactive subcategory and *Wrangler*, *MTL* from the proactive subcategory are in this group. *MRRL* is appropriate for an unknown heterogeneous environment because no prior knowledge is required about environmental characteristics. *Wrangler* is not appropriate for environments that change continuously, as it builds a separate model for each node and workload. On the other hand, *MTL* is appropriate for an environment with related tasks and requires less training data. The preferred choice for this environment is *MTL* because it requires less training time than *Wrangler*.
4. **Heterogeneous with large-size jobs**: *PFSE* is the only job scheduler that is specifically for this situation. It is designed for the identification of stragglers with high accuracy in data-intensive cloud computing

Table 4 Guidelines for job scheduler selection for mitigating stragglers

Job scheduler	Environment	Job size	Job type	Input size	Cluster size (nodes)	Flexibility	Scalability	Complexity
LATE	Homogenous and static	Small	Single type	Small	9	No	Low	Low
SAMR	Heterogenous and dynamic	Small	Reduce tasks dominate	Small	6	Yes	Low	Medium
Mantri	Heterogenous/known	Small	Different types	Large	40	Yes	Low	Medium
CREST	Heterogenous	Small	Different types	Large	10	Yes	Low	High
ESAMR	Heterogenous and dynamic	Small	Different types	Small	6	Yes	Low	High
MRRL	Heterogenous/unknown	Medium	Different types	N/A	N/A	Yes	Medium	Medium
Tolhit	Heterogenous	Small	Different types	Small	5	Yes	Low	High
PFSE	Heterogenous	Large	Single type	Small	N/A	Yes	Medium	Medium
Dolly	Homogenous	Small	Interactive jobs	Small	150	Yes	Medium	High
Wrangler	Heterogenous	Medium	Repetitive jobs	High	50	Yes	High	High
MTL	Heterogenous	Medium	Related tasks	Small	50	Yes	High	High

environments with a single type of job. It uses completed task information to estimate execution time; therefore, by increasing the number of completed tasks, its accuracy is increased.

7.2 Job scheduler selection for improving data locality

We present guidelines to select the appropriate job scheduler to improve data locality in each environment. Table 5 presents some environmental features that we consider for choosing a job scheduler for improving data locality for map tasks, reduce tasks, and both map and reduce tasks. We have divided job schedulers among these four environmental categories as follows:

1. **Homogenous environment**: *Delay*, *Matchmaking*, *NKS*, and *Joint* from data locality for map tasks, and *LARTS* from data locality for reduce tasks are appropriate for this environment. *Delay* is suitable for small jobs in a homogenous environment because it improves response times for small jobs by a factor of 5 in a multi-user workload and can double throughput in an I/O-heavy workload. However, it is not effective if a significant fraction of tasks is much longer than the average job, or if there are limited slots per node. *Matchmaking* is similar to *Delay*, but it does not need to tune delay time. *NKS* is very suitable for homogenous environments with high network loads because it aims to reduce network load. *Joint* is not appropriate in heterogeneous network environments because it does not consider different link bandwidths and the various structures of the connections between nodes. *LARTS* decreases the network traffic significantly, so it is suitable for an environment with high network traffic. With this in mind, *NKS*, *LARTS* could be considered preferred options.
2. **Heterogeneous with small size jobs**: *ILA*, *HPSO*, and *HybsMRP* from data locality for map tasks, and *LONARS* from data locality for reduce tasks are suitable for this environment. *ILA* performs better for small jobs because the degree of parallelism (and hence the performance degradation due to the increased parallel degree) is limited by the number of tasks in small jobs. *HPSO* is a scheduler that uses a prefetch mechanism; therefore, it is appropriate for I/O-bound jobs and heterogeneous environments. *LONARS*

Table 5 Guidelines for job scheduler selection for improving data locality

Job scheduler	Environment	Job size	Job type	Input size	Cluster size (nodes)	Flexibility	Scalability	Complexity
Delay	Homogenous	Small	I/O-heavy workload	Small	100	No	Medium	Low
Matchmaking	Homogenous	Small	Different job types	Small	30	No	Low	Low
NKS	Homogenous with high network load	Small	Different job types	Small	10	Yes	Low	High
Maestro	Both	Small	I/O-bound	Small	100	Yes	Medium	Medium
ILA	Heterogeneous	Small	CPU-bound and I/O-bound	Small	72	Yes	Medium	High
HPSO	Heterogeneous	Small	I/O-bound	N/A	N/A	Yes	Low	Medium
Joint	Homogenous	Small	N/A	N/A	200	No	Medium	Medium
ELRSA	Heterogenous	Various	I/O-bound, CPU-bound	Large	28	Yes	Medium	Medium
Predictive	Heterogenous	Large	All job types	N/A	5	Yes	Medium	High
HybSMRP	Heterogenous	Small	Different job types	Large	20	Yes	Low	Low
LARTS	Homogenous	Small	Different job types	Small	14	Yes	Low	Medium
COGRS	Both	Small	Different job types	Small	14	Yes	Low	High
LONARS	Heterogenous	Small	Different job types	Small	12	Yes	Low	High
Online simulation	Both	Small	Single job type	Small	3	No	Low	High
Multithreading locality	Both	Large	Different job types	Large	N/A	Yes	High	High
JOSS	Homogenous	Various	Different job types	Small	30	Yes	Medium	Medium
DGIA	Heterogenous	Large	Different job types	High	1000	Yes	High	Medium

Table 6 Guidelines for job scheduler selection for improving resource utilization

Job scheduler	Environment	Job size	Job type	Input Size	Cluster size (nodes)	Flexibility	Scalability	Complexity
Fair	Homogenous	Small	Different	Small	5	No	Low	Low
Capacity	Homogenous	Small	Different	Small	5	Yes	Low	Medium
COOMR	Homogenous	Small	Data-intensive with heavy reduce phase	High	25	Yes	Low	High
DynMR	Homogenous	Small	Different	Small	4	Yes	Low	Medium
RAS	Both	Different	Different kinds of workloads	Small	5	Yes	Medium	Medium
RAAS	Both	Different	Multi-job workloads	High	22	Yes	Medium	Medium
CASH	Heterogeneous	Small	Different	Small	4	Yes	Low	Medium
COSHH	Heterogeneous	Large	Different	High	6	Yes	Medium	High
PRISM	Both	Same size	Wide range of workloads	Small	10	Yes	Low	High
WCRA	Heterogeneous	Different	Different	Small	6	Yes	Medium	High
JAS	Heterogeneous	Different	Various job workloads	High	7	Yes	Medium	Medium
DGNS	Heterogeneous	Different	Different	Small	15	Yes	Medium	High
EMRSAY	Heterogeneous	Large	Different	High	5	Yes	Medium	Medium

generates significant overhead for large jobs and has relatively high complexity.

3. **Heterogeneous with large size jobs**: *Predictive* and *ELRSA* from data locality for map tasks, and *JOSS*, *DGIA* from data locality for jobs are well-suited for this environment. *Predictive* uses a linear regression model to predict the task execution time, so has better prediction accuracy for larger jobs. Therefore, it is better to use this method for large size jobs in a heterogeneous environment. *ELRSA* is designed for a heterogeneous cluster environment that can support various workloads and I/O-bound and CPU-bound jobs. *JOSS* is ideal for different job sizes and varied workloads, and it can be used when the jobs are not small. *DGIA* can reduce the number of deadline misses for large workloads.
4. **Combined environment (Homogenous and Heterogeneous)**: *Maestro* from data locality for map tasks, *CoGRS* from data locality for reduce tasks, and *Online simulation*, *MultiThreading Locality* from data locality for jobs appear to work well in these environments. *Maestro* can be applied to both homogenous and heterogeneous environments; it performs best for I/O-bound jobs. However, this method is not suitable for unknown environments because we require knowledge of how data replication is performed. *COGRS* has demonstrated excellent performance on a dedicated homogenous cluster and shared heterogeneous cloud, and it is useful for reduce tasks with high partition skew to reduce network traffic. It performs best when map outputs are divided among reduce tasks non-uniformly. A limitation of the Task Predictor in the *Online simulation* method is that it predicts execution time based on the completion time of previous tasks from the same job. Therefore, it is best used for a single type of job so that execution time predictions have high accuracy, and it is more suitable for small clusters as it generates significant overhead for large clusters.

7.3 Job scheduler selection for improving resource utilization

We present guidelines to select an appropriate job scheduler to improve resource utilization in each environment. Table 6 presents some environmental features that we consider for choosing the best job scheduler for each group: fair distribution and resource-aware. We have distributed job schedulers among five environmental categories as follows:

1. **Homogenous environment**: *Capacity*, *COOMR*, and *DynMR* are appropriate for this environment. *COOMR* is ideal for data-intensive applications with heavy reduce phases and large clusters in a homogenous environment as it focuses on reduce tasks and has high scalability.
2. **Heterogeneous with small size jobs**: *CASH* is suitable for heterogeneous, dynamic, and shared environments because it supports dynamic changes in the availability of resources.

Table 7 A brief of scheduling techniques in different hadoop types

Hadoop system type	Job scheduling techniques	Best for situation	Implementation	Simulation	Formally study
Homogenous	LATE (straggler detection) delay, matchmaking, NKS, joint, and LARTS (data locality) capacity, CooMR, DynMR (resource utilization)	Capacity is common for resource utilization, Delay is common for data locality, NKS, LART are suitable for environment with high network load, COOMR for data-intensive application and large cluster	LATE, Delay, Matchmaking, NKS, LARTS, Capacity, CooMR, DynMR	Joint	N/A
Heterogeneous with small size jobs	SAMR, Mantri, CREST, ESAMR, Tolhit, and Dolly (Straggler detection) ILA, HPSO, and HybsMRP, and LONARS (Data locality) CASH	Dolly for interactive jobs and common for straggler prediction, ESAMR is common for straggler detection, HPSO is suitable for I/O- bound jobs, CASH for a dynamic environment	SAMR, ESAMR, ILA, HybsMRP, LONARS, CASH	Mantri, CREST, Tolhit, Dolly	HPSO
Heterogeneous with medium size jobs	MRRL, Wrangler, and MTL (Straggler detection)	MRRL for the unknown environment, MTL has the best accuracy for straggler prediction	Wrangler	N/A	MRRL, MTL
Heterogeneous with large size jobs	PFSE (Straggler detection) predictive and ELRSA, JOSS, and DGIA (Data locality) COSHH, EMRSAY (Resource utilization)	PFSE for data-intensive cloud computing environments, COSHH for large cluster	ELRSA, JOSS, DGIA, EMRSAY	PFSE, COSHH	Predictive
Heterogeneous with different size jobs	Fair, WCRA, and JAS (Resource utilization)	Fair is common, JAS for a multiuser environment, WCRA for a small cluster	Fair, WCRA	JAS	N/A
Combined system (Homogeneous and Heterogeneous)	Maestro, CoGRS, Online simulation, and MultiThreading Locality (Data locality) RAS, RAAS, and PRISM (Resource utilization)	Maestro is suitable for I/O-bound jobs, PRISM for repeated execution	Maestro, COGRS, RAS, RAAS, PRISM	Online simulation, multithreading locality	N/A

3. **Heterogeneous with large size jobs**: *COSHH* and *EMRSAY* are appropriate for large size jobs and large clusters in a heterogeneous environment because *COSHH* reduces communication cost and overhead, and *EMRSAY* saves energy.
4. **Heterogeneous with different size jobs**: *Fair* can be used for both small and large size jobs. *WCRA* is suitable for various workloads on small clusters in a heterogeneous environment as it considers workload status before tableeous and multi-user environment.
5. **Combined environment (Homogenous and Heterogeneous)**: *RAS* improves resource utilization in both homogenous and heterogeneous environments when different kinds of workloads run on the cluster. *RAAS* is appropriate for jobs that run periodically on different size data with uniform characteristics in both environments. *PRISM* is ideal for an environment where jobs are executed repeatedly with the same input size because the accuracy of the job profilers can be improved over time.

A brief of job scheduling techniques in various Hadoop system types is presented in Table 7 which summarizes Hadoop system features with the most appropriate job scheduler techniques and the best situation that users can use. In addition, it shows which schedulers have been implemented or simulated, or only theoretically studied.

8 Conclusion and future work

We have provided an overview of different MapReduce job schedulers in Hadoop and have classified them into three groups based on approaches to improve performance:(1) Mitigating stragglers, (2) Improving data locality, (3) Improving resource utilization. The first category uses two strategies, reactive and proactive, in reducing the adverse effects on performance. A proactive strategy is typically better than a reactive strategy because it predicts and prevents the occurrence of a straggler; therefore, it avoids resource wastage. The second category is for improving data locality and is divided into three subcategories: (1) Data locality for map tasks, (2) Data locality for reduce tasks, and (3) Data locality for both map and reduce tasks. Improving data locality for reduce tasks is more complex than data locality for map tasks because the output of the map tasks serves as input data for the associated reduce tasks. As a result, prior information about the location of input data for the reduce tasks is not available. The last group is for improving resource utilization, divided into two subgroups: (1) Fair resource distribution, and (2) Resource-aware mechanisms. The first group does not use information about the characteristics of jobs and available resources; therefore, the schedulers in this group do not generate significant overhead. Using different methods, the second group classifies jobs into two groups: I/O-bound and CPU-bound, and assigns them to the appropriate nodes.

We presented guidelines to select a job scheduler based on scheduler features and the performance impact in different operational environments. Amongst reactive techniques, *ESAMR* is preferred for straggler identification, and *Tolhit* is preferred for selecting target nodes on which to launch backup tasks. Amongst proactive techniques, *Wrangler* and *MTL* have high accuracy and reliability in straggler prediction; however, *Wrangler* requires a longer training time than *MTL*; therefore, the preferred existing proactive technique is *MTL*. *CREST* and *Wrangler* have the best impact on performance. *Maestro*, *HPSO*, and *ELRSA* have the highest data locality rate for map tasks, and *LARTS* has the maximum data locality rate for reduce tasks. *Predictive* and *COGRS* have the greatest impact on performance.

All of these job schedulers focus on one performance aspect. In some cases, optimizing one performance metric can result in significant degradation in another metric. However, no scheduling algorithm considers all of these performance metrics. One of our future goals is to design a hybrid job scheduler that considers a combination of these performance aspects; for example, *JASL* is a job scheduler that considers both data locality with resource utilization; the result can be a superior impact on Hadoop performance.

References

1. Usama, M., Liu, M., Chen, M.: Job schedulers for big data processing in Hadoop environment: testing real-life schedulers using benchmark programs. Digit. Commun. Netw. **3**, 260–273 (2017)
2. Gautam, J. V., Prajapati, H. B., Dabhi, V. K. & Chaudhary, S.: A survey on job scheduling algorithms in Big data processing. In: Proceedings of 2015 IEEE International Conference on Electrical, Computer and Communication Technologies, ICECCT 2015 (2015). https://doi.org/10.1109/ICECCT.2015.7226035
3. Abdallat, A.A., Alahmad, A.I., Amimi, D.A.A., AlWidian, J.A.: Hadoop mapreduce job scheduling algorithms survey and use cases. Mod. Appl. Sci. **13**, 38 (2019)
4. Kalia, K., Gupta, N.: Analysis of hadoop mapreduce scheduling in heterogeneous environment. Ain Shams Eng. J. **12**, 1101–1110 (2021)
5. Zaharia, M., Konwinski, A., Joseph, A. D., Katz, R. & Stoica, I.: Improving mapreduce performance in heterogeneous environments. In: Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008. p. 29–42 (2019)
6. Chen, Q., Zhang, D., Guo, M., Deng, Q. & Guo, S.: SAMR: a self-adaptive mapreduce scheduling algorithm in heterogeneous environment. In: Proceedings—10th IEEE International Conference on Computer and Information Technology, CIT-2010, 7th IEEE International Conference on Embedded Software and Systems, ICESS-2010, ScalCom-2010. pp. 2736–2743 (2010). https://doi.org/10.1109/CIT.2010.458
7. Ananthanarayanan, G. et al.: Reining in the outliers in map-reduce clusters using mantri. In: Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010. p. 265–278 (2019)
8. Lei, L., Wo, T. & Hu, C.: CREST: towards fast speculation of straggler tasks in mapreduce. In: Proceedings—2011 8th IEEE International Conference on e-Business Engineering, ICEBE 2011. p. 311–316 (2011)
9. Sun, X., He, C. & Lu, Y.: ESAMR: an enhanced self-adaptive mapreduce scheduling algorithm. In: Proceedings of the International Conference on Parallel and Distributed Systems—ICPADS. p. 148–155 (2012). https://doi.org/10.1109/ICPADS.2012.30
10. Naik, N.S., Negi, A., Sastry, V.N.: Performance improvement of mapreduce framework in heterogeneous context using reinforcement learning. Procedia Comput. Sci. **50**, 169–175 (2015)
11. Brahmwar, M., Kumar, M., Sikka, G.: Tolhit—a scheduling algorithm for hadoop cluster. Procedia Comput. Sci. **89**, 203–208 (2016)
12. Ibrahim, I. A. & Bassiouni, M.: Improving mapreduce performance with progress and feedback based speculative execution. In: Proceedings—2nd IEEE International Conference on Smart Cloud, SmartCloud 2017. p. 120–125 (2017). https://doi.org/10.1109/SmartCloud.2017.25
13. Ananthanarayanan, G., Ghodsi, A., Shenker, S. & Stoica, I.: Effective straggler mitigation: attack of the clones. In: Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2013. p. 185–198 (2013)
14. Yadwadkar, N. J., Ananthanarayanan, G. & Katz, R.: Wrangler: predictable and faster jobs using fewer resources. In: Proceedings of the 5th ACM Symposium on Cloud Computing, SOCC 2014 (2014). https://doi.org/10.1145/2670979.2671005
15. Li, Y., Yang, Q., Lai, S., Li, B.: A new speculative execution algorithm based on C4.5 decision tree for hadoop. In: Wang, H., Qi, H., Che, W., Qiu, Z., Kong, L., Han, Z., Lin, J., Lu, Z. (eds.) International Conference of Young Computer Scientists, Engineers and Educators, pp. 284–291. Springer, Berlin (2015)
16. Yadwadkar, N.J., Hariharan, B., Gonzalez, J.E., Katz, R.: Multi-task learning for straggler avoiding predictive job scheduling. J. Mach. Learn. Res. **17**, 1–37 (2016)
17. Zaharia, M. et al.: Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling. In: Proceedings of the 5th European conference on Computer systems. p. 265 (2010). https://doi.org/10.1145/1755913.1755940

18. He, C., Lu, Y. & Swanson, D.: Matchmaking: a new mapreduce scheduling technique. In: Proceedings—2011 3rd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2011. p. 40–47 (2011). https://doi.org/10.1109/CloudCom.2011.16
19. Zhang, X., Zhong, Z., Feng, S., Tu, B. & Fan, J.: Improving data locality of mapreduce by scheduling in homogeneous computing environments. In: Proceedings—9th IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2011. 2, p. 120–126 (2011).
20. Ibrahim, S. et al.: Maestro: replica-aware map scheduling for mapreduce. In: Proceedings—12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid 2012. p. 435–442 (2012). https://doi.org/10.1109/CCGrid.2012.122
21. Bu, X., Rao, J. & Xu, C. Z.: Interference and locality-aware task scheduling for mapreduce applications in virtual clusters. In: HPDC 2013—Proceedings of the 22nd ACM International Symposium on High-Performance Parallel and Distributed Computing. p. 227–238 (2013). https://doi.org/10.1145/2462902.2462904
22. Tamil Selvan, S., Dhamotharan, K.A., Saravanan, G., Karunamoorthi, R.: Investigation analysis on data prefetching and mapreduce techniques for user query processing. Int. J. Sci. Technol. Res. **9**, 2185–2189 (2020)
23. Wang, W., Ying, L.: data locality in mapreduce: a network perspective. Perform. Eval. **96**, 1–11 (2016)
24. Bibal Benifa, J.V., Dejey, D.: Performance improvement of mapreduce for heterogeneous clusters based on efficient locality and replica aware scheduling (ELRAS) strategy. Wirel. Pers. Commun. **95**, 2709–2733 (2017)
25. Merabet, M., Benslimane, S.M., Barhamgi, M., Bonnet, C.: A predictive map task scheduler for optimizing data locality in mapreduce clusters. Int. J. Grid High Perform. Comput. **10**, 1–14 (2018)
26. Gandomi, A., Reshadi, M., Movaghar, A., Khademzadeh, A.: HybSMRP: a hybrid scheduling algorithm in hadoop mapreduce framework. J. Big Data (2019). https://doi.org/10.1186/s40537-019-0253-9
27. Rehman, S. Locality-Aware Reduce Task Scheduling for MapReduce Mohammad Hammoud and Presented By: Problem At Hand. 1–14.
28. Hammoud, M., Rehman, M. S. & Sakr, M. F.: Center-of-gravity reduce task scheduling to lower MapReduce network traffic. In: Proceedings—2012 IEEE 5th International Conference on Cloud Computing, CLOUD 2012. p. 49–58 (2012). https://doi.org/10.1109/CLOUD.2012.92
29. Arslan, E., Shekhar, M. & Kosar, T.: Locality and network-aware reduce task scheduling for data-intensive applications. In: Proceedings of DataCloud 2014: 5th International Workshop on Data Intensive Computing in the Clouds—Held in Conjunction with SC 2014: The International Conference for High Performance Computing, Networking, Storage and Analysis. p. 17–24 (2014). https://doi.org/10.1109/DataCloud.2014.10
30. Wang, G., Khasymski, A., Krish, K. R. & Butt, A. R.: Towards improving mapreduce task scheduling using online simulation based predictions. In: Proceedings of the International Conference on Parallel and Distributed Systems—ICPADS. p. 299–306 (2013). https://doi.org/10.1109/ICPADS.2013.50
31. Suresh, S., Gopalan, N.P.: An optimal task selection scheme for hadoop scheduling. IERI Procedia **10**, 70–75 (2014)
32. Adhianto, L., et al.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**, 685–701 (2010)
33. Lee, M.C., Lin, J.C., Yahyapour, R.: Hybrid job-driven scheduling for virtual mapreduce clusters. IEEE Trans. Parallel Distrib. Syst. **27**, 1687–1699 (2016)
34. Joseph, J.L., Lin, M.A.C., Lin, J., Lin, C.: Joint deadline-constrained and influence-aware design for allocating mapreduce jobs in cloud computing systems. Clust. Comput. **22**(3), 6963–6976 (2018)
35. Goals, F. S. et al. Hadoop fair scheduler design document. p. 1–11 (2010).
36. Chen, J., Wang, D., Zhao, W.: A task scheduling algorithm for Hadoop platform. J. Comput. **8**, 929–936 (2013)
37. Li, X., Wang, Y., Jiao, Y., Xu, C. & Yu, W.: CooMR: cross-task coordination for efficient data management in mapreduce programs. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC (2013). https://doi.org/10.1145/2503210.2503276
38. Sagar, A., Moni, R.V.: DynMR: a dynamic slot allocation framework for mapreduce clusters in big data management using DHSA and SEPB. Int. J. Comput. Tech. **2**, 142–155 (2017)
39. Yong, M., Garegrat, N. & Mohan, S. Towards a resource aware scheduler in hadoop. Proc. ICWS 1–10 (2009).
40. Polo, J. et al. Resource-Aware Adaptive Scheduling for MapReduce Clusters To cite this version: HAL Id: hal- 01597795 Resource-aware Adaptive Scheduling for MapReduce Clusters. 0–20 (2017).
41. Cassales, G.W., Charão, A.S., Pinheiro, M.K., Souveyet, C., Steffenel, L.A.: Context-aware scheduling for Apache Hadoop over pervasive environments. Procedia Comput. Sci. **52**, 202–209 (2015)
42. Rasooli, A., Down, D.G.: COSHH: a classification and optimization based scheduler for heterogeneous Hadoop systems. Future Gener. Comput. Syst. **36**, 1–15 (2014)
43. Zhang, Q., Zhani, M.F., Yang, Y., Boutaba, R., Wong, B.: PRISM: fine-grained resource-aware scheduling for mapreduce. IEEE Trans. Cloud Comput. **3**, 182–194 (2015)
44. Divya, M. & Annappa, B.: Workload characteristics and resource aware Hadoop scheduler. In: 2015 IEEE 2nd International Conference on Recent Trends in Information Systems, ReTIS 2015—Proceedings. p. 163–168 (2015). https://doi.org/10.1109/ReTIS.2015.7232871
45. Hsieh, S.Y., et al.: Novel scheduling algorithms for efficient deployment of mapreduce applications in heterogeneous computing environments. IEEE Trans. Cloud Comput. **6**, 1080–1095 (2018)
46. Chen, C.T., Hung, L.J., Hsieh, S.Y., Buyya, R., Zomaya, A.Y.: Heterogeneous job allocation scheduler for Hadoop mapreduce using dynamic grouping integrated neighboring search. IEEE Trans. Cloud Comput. **8**, 193–206 (2020)
47. Pandey, V.: A heuristic method towards deadline-aware energy-efficient mapreduce scheduling problem in Hadoop YARN. Clust. Comput. (2020). https://doi.org/10.1007/s10586-020-03146-7

Rana Ghazali is currently a Ph.D. student in computer engineering (software discipline) at the Islamic at Azad University North Tehran Branch, Iran and she is a visiting student at McMaster University, Canada from March 2020 until now. Her main research interests include Big Data, MapReduce job scheduler, performance evaluation, machine learning.

Sahar Adabi is an assistant professor with the North Tehran Branch of IAU. She's participated as a software architect and senior consultant in several large-scale software software-intensive projects. Her main research interests include software architecture, scheduling, and resource management in dynamic and massively parallel systems, IoT, pervasive computing, and behavioral modeling and prediction of mobile IoT/Fog/Cloud-based systems.

Douglas G. Down received his B.A.Sc. and M.A.Sc. degrees from the University of Toronto (1986 and 1990) and his Ph.D. from the University of Illinois at Urbana– Champaign (1994). His interests lie in performance evaluation and resource allocation in distributed computer systems. He is currently the Academic Director of the Computing Infrastructure Research Centre at McMaster University, Canada.

Ali Movaghar is a Professor in the Department of Computer Engineering at Sharif University of Technology in Tehran, Iran, and has been on the Sharif faculty since 1993. He received his B.S. degree in Electrical Engineering from the University of Tehranin1977, and M.S. and Ph.D. degrees in Computer, Information, and Control Engineering from the University of Michigan, Ann Arbor, in 1979 and1985, respectively.Hevisitedthe InstitutNationalde Recherche en Informatique et en Automatique in Paris, France, and the Department of Electrical Engineering and Computer Science at the University of California, Irvine in 1984 and 2011, respectively, worked at AT&T Information Systems in Naperville, IL in 1985–1986, and taught at the University of Michigan, Ann Arbor in 1987–1989. His research interests include performance/dependability modeling and formal verification of wireless networks and distributed real-time systems. He is a senior member of the IEEE and the ACM.