Dynamic Job Ordering and Slot Configurations for MapReduce Workloads

Shanjiang Tang, Bu-Sung Lee, and Bingsheng He

Abstract—MapReduce is a popular parallel computing paradigm for large-scale data processing in clusters and data centers. A MapReduce workload generally contains a set of jobs, each of which consists of multiple map tasks followed by multiple reduce tasks. Due to 1) that map tasks can only run in map slots and reduce tasks can only run in reduce slots, and 2) the general execution constraints that map tasks are executed before reduce tasks, different job execution orders and map/reduce slot configurations for a MapReduce workload have significantly different performance and system utilization. This paper proposes two classes of algorithms to minimize the *makespan* and the *total completion time* for an offline MapReduce workload. Our first class of algorithms focuses on the job ordering optimization for a MapReduce workload under a given map/reduce slot configuration. In contrast, our second class of algorithms considers the scenario that we can perform optimization for map/reduce slot configuration for a MapReduce workload. We perform simulations as well as experiments on Amazon EC2 and show that our proposed algorithms produce results that are up to $15 \sim 80$ percent better than currently unoptimized Hadoop, leading to significant reductions in running time in practice.

Index Terms—MapReduce, Hadoop, flow-shops, scheduling algorithm, job ordering

1 Introduction

MapReduce is a widely used computing model for large scale data processing in cloud computing. A MapReduce *job* consists of a set of map and reduce *tasks*, where reduce tasks are performed after the map tasks. Hadoop [2], an open source implementation of MapReduce, has been deployed in large clusters containing thousands of machines by companies such as Amazon and Facebook. In those cluster and data center environments, MapReduce and Hadoop are used to support batch processing for jobs submitted from multiple users (i.e., MapReduce workloads). Despite many research efforts devoted to improve the performance of a single MapReduce job (e.g., [3], [11]), there is relatively little attention paid to the system performance of MapReduce workloads. Therefore, this paper tries to improve the performance of MapReduce workloads.

Makespan and *total completion time (TCT)* are two key performance metrics. Generally, *makespan* is defined as the time period since the start of the first job until the completion of the last job for a set of jobs. It considers the computation time of jobs and is often used to measure the performance and utilization efficiency of a system. In contrast, *total completion time* is referred to as the sum of completed time periods for all jobs since the start of the first job. It is a generalized makespan with queuing time (i.e., waiting time) included. We can use it to measure the satisfaction to the system from a single job's perspective through dividing the total completion time by the number of jobs (i.e., average completion time). Therefore, in this paper, we aim to optimize these two metrics.

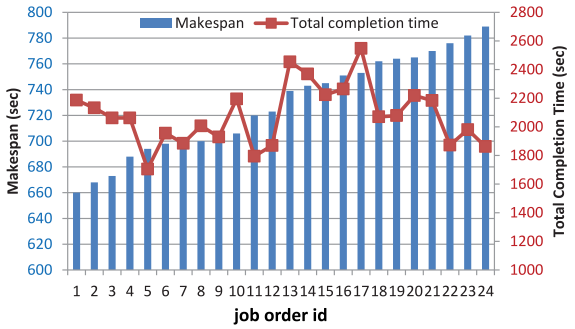
We consider the production MapReduce workloads whose jobs run periodically for processing new data. The default FIFO scheduler is often adopted in order to minimize the overall execution time [41]. The analysis is generally performed offline to optimize the execution for such production workloads. There are a surge amount of optimization approaches on that. For example, Rasmussen et al. [33] and Jiang et al. [21] considers the low-level I/O-efficient optimization. Agrawal et al. [8] and Nykiel et al. [27] share the operation by eliminating redundant data access and computation.

In this paper, we target at one subset of production MapReduce workloads that consist of a set of independent jobs (e.g., each of jobs processes distinct data sets with no dependency between each other) with different approaches. For dependent jobs (i.e., MapReduce workflow), one MapReduce can only start only when its previous dependent jobs finish the computation subject to the input-output data dependency. In contrast, for independent jobs, there is an overlap computation between two jobs, i.e., when the current job completes its map-phase computation and starts its reduce-phase computation, the next job can begin to perform its map-phase computation in a pipeline processing mode by possessing the released map slots from its previous job. Particularly, as shown in Fig. 1b in Section 2.1, different job submission orders result in varied computation overlaps, and in turn the different cluster utilizations and performance.

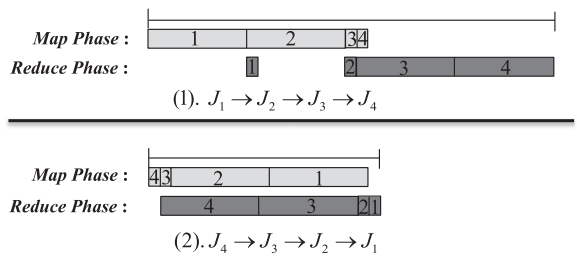
Moreover, in Hadoop MRv1, it abstracts the cluster resources into slots (e.g., map slots and reduce slots). Due to varied slot demands for map and reduce tasks, different map/reduce slot configurations can also have significantly different performance and system utilization. Particularly,

- *S.J. Tang is with the School of Computer Science and Technology, Tianjin University, China. E-mail: tashj@tju.edu.cn.*
- *B.-S. Lee and B.S. He are with the School of Computer Engineering, Nanyang Technological University, Singapore. E-mail: {ebslee, bshe}@ntu.edu.sg.*

Manuscript received 31 Oct. 2014; revised 6 Apr. 2015; accepted 12 Apr. 2015. Date of publication 23 Apr. 2015; date of current version 10 Feb. 2016. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TSC.2015.2426186



(a) The experimental results for a batch of 4 jobs run on the Amazon EC2 Hadoop cluster under all $4! = 24$ job execution orders.



(b) MapReduce execution flow for different job orders.

Fig. 1. Performance comparison for a batch of jobs under different job orders in FIFO scheduling.

the importance and challenges of map/reduce slot configuration optimization are motivated in Section 2.2.

With these observations, we try to improve the performance for MapReduce workloads with job ordering and slot configuration optimization approaches.

Job ordering optimization. We start by describing the job ordering algorithm *MK_JR* based on *Johnson's Rule* [22] for makespan optimization. It turns out to be equivalent to the two-stage flow shop problem when there are one map slot and one reduce slot only [41]. The *Johnson's Rule* [22] can produce the optimal job order for makespan in this case. When it comes to the general case where there are arbitrary number of map and reduce slots. Minimizing the makespan in this case is $\mathcal{NP}$-hard. We show that *MK_JR* produces a $1+\delta$ approximation to the minimum makespan, where $\delta < 1$ is the ratio of sum of the maximum map and reduce task size, to the sum of all the task sizes. Since the processing time of a single map/reduce task is typically small in practice compared to the overall execution time of a MapReduce workload, δ is usually a very small quantity [44]. Moreover, we see that there is a significant trade-off between the makespans and total completion times of the orderings, as explained in Section 2.1. We then further propose a bi-criteria algorithm *MK_TCT_JR* to optimize both makespan and total completion time together.

Having proposed job ordering algorithms that optimize the makespan and total completion time, we also show that they are *stable*, i.e., the optimized orders produced by job ordering algorithms do not change even if some MapReduce servers fail at execution time (See Theorem 3 in Section 5.3).

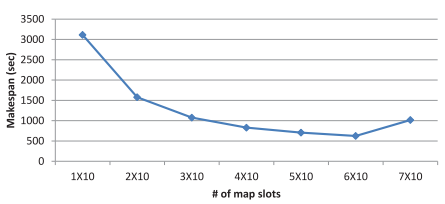
Slot configuration optimization. Moreover, the slot configuration can have a significant impact on performance for MapReduce workloads. Our motivating experiments in Section 2.2 show that there are 399 percent performance difference between the optimal and the worst-case slot configurations (See Fig. 2a). We propose several enumeration algorithms for map/reduce slot configuration optimization with regard to the makespan and total completion time of a MapReduce workload.

Having proposed enumeration algorithms for map/reduce slot configuration optimization, we also show that there is a *proportional* relationship for the optimized map/reduce slot configurations for any two different sizes of total slots (See **PCP** in Section 6.3). It is important to address the time efficiency problem of the proposed enumeration algorithms for a large-size number of total slots.

Experimental results. We evaluate our algorithms using both testbed workloads and synthetic Facebook workloads. Experiments show that, 1). for the makespan, the job ordering optimization algorithm achieve an approximately 14-36 percent improvement for testbed workloads, and 10-20 percent improvement for Facebook workloads. In contrast, with the map/reduce slot configuration algorithm, there are about 50-60 percent improvement for testbed workloads and 54-80 percent makespan improvement for Facebook workloads; 2). for the total completion time, there are nearly $5\times$ improvement with the bi-criteria job ordering algorithms and $4\times$ improvement with the bi-criteria map/reduce slot configuration algorithm, for Facebook workloads that contain many small-size jobs,

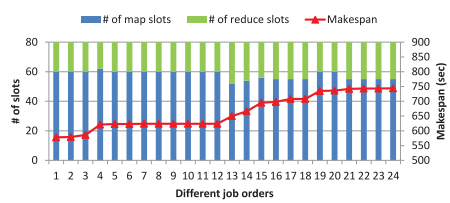
The main contributions of this paper are summarized as follows:

- Have a theoretical study on the *Johnson's Rule*-based heuristic algorithm for makespan, including its approximation ratio, upper bound and lower bound makespan.
- Propose a bi-criteria heuristic algorithm to optimize makespan and total completion time simultaneously, observing that there is a tradeoff between makespan



(a) The results for all possible map/reduce slot configurations in practice under an arbitrarily given job submission order. The optimal number of map slots occurs at Point 6×10 with the minimum makespan of 624.

(b) The results for all $4! = 24$ possible job execution orders under the optimal map/reduce slot configuration, sorted by makespan.



(c) The optimal map/reduce slot configuration for all $4! = 24$ possible job execution orders with regard to the makespan minimization, sorted by makespan.

Fig. 2. The motivating example of a MapReduce workload consisting of four jobs under FIFO scheduler with a total number of 80 slots.

and total completion time from Fig. 1a. Moreover, we show that the optimized job order produced by the proposed orderings algorithms does not need to change (i.e., *stable*) in the face of server failures via theoretical analysis.
- Propose slot configuration algorithms for makespan and total completion time. We also show that there is a *proportional* feature for them, which is very important and can be used to address the time efficiency problem of proposed enumeration algorithms for a large size of total slots.
- Perform extensive experiments to validate the effectiveness of proposed algorithms and theoretical results.

The rest of the paper is organized as follows. Section 2 motivates the importance and challenges of job ordering optimization and slot configuration optimization for the performance of MapReduce workloads. Section 3 reviews the related work. Section 4 formulates problems and gives performance model for makespan and total completion time. We present our job ordering algorithms in Section 5. Section 6 gives our map/reduce slot configuration optimization algorithms. In Section 7, we give performance evaluation to validate the optimization effectiveness of these algorithm experimentally. Finally, Section 8 concludes the paper.

2 Motivation

In this section, we show the importance and challenge of job ordering optimization as well as map/reduce slot configuration optimization by giving motivating examples experimentally.

2.1 Motivation for Job Ordering Optimization

To motivate the importance of job ordering optimization for MapReduce workloads on performance, we ran a testbed workload consisting of four jobs ($J_1 \sim J_4$) from Table 2 of Section 7.1 in a Amazon EC2 Hadoop cluster configured with map slots of 57 and reduce slots of 19. We do so by comparing the performance for all possible job submission orders.

Fig. 1a shows the results for all $4! = 24$ job submission orders, sorted by *makespan*. The x-axis shows the job order id for all job orders. We observe that there is a 20 percent (130 second) difference between the best and worst ordering. Depending on the characteristics of the workload, this difference can be even greater. Fig. 1b shows another set of jobs where the makespan can differ by nearly 100 percent. This is due to the non-overlap computation constraint between map and reduce tasks of a MapReduce job, resulting in different resource utilizations for map/reduce slots under different job submission orders for batch jobs, as illustrated in Fig. 1b. However, the job ordering optimization for MapReduce workloads is important and challenging, due to the following facts: (i). There is a strong data dependency between the map tasks and reduce tasks of a job, i.e., reduce tasks can only perform after the map tasks, (ii). map tasks have to be allocated with map slots and reduce tasks have to be allocated with reduce slots, (iii). Both map slots and reduce slots are limited computing resources, configured by Hadoop administrator in advance [26]. Because of these, different job submission orders will result in different resource utilizations for map and reduce slots and in turn different performance (i.e., makespan) for batch jobs. Moreover, it is worth mentioning that the computation for each reduce task consists of three sub-phases, namely, *shuffle*, *sort* and *aggregation*. The shuffle phase computation of reduce tasks can start earlier before all map tasks are completed, whereas other two sub-phases cannot. It means that only sort and aggregation phases of reduce tasks cannot overlap with all map tasks. Moreover, a MapReduce job execution generally exhibits the multiple waves in its map and reduce phase. Only the first wave of reduce tasks can overlap their data shuffling with map task computation. For other remaining reduce tasks, they can only start shuffling after all map tasks are completed. In summary, we cannot avoid the strict (or partial) non-overlap data dependency between map and reduce tasks. In our testbed experiment above, we keep the default Hadoop configuration, which enables reduce tasks to start running when there are 5 percent of map tasks completed. The different job ordering results are just to be an effective validation for this point.

Moreover, in Fig. 1a, the curve running through the middle of the figure gives the corresponding *total completion time* (i.e., the sum of the completion times of all the jobs) for different orders. We see that there is a significant trade-off between the makespans and total completion times of the orderings. In fact, job order 5 has a makespan which is close to the optimal, but its total completion time is much better than the total completion time of the minimum makespan order. It indicates that there is a need to optimize both makespan and total completion time together.

2.2 Motivation for Slot Configuration Optimization

To motivate the importance of optimization on map/reduce slot configuration, we perform a simulation experiment with a testbed MapReduce workload consisting of four jobs, assuming that a cluster consisting of 10 slave nodes each configured with eight slots, i.e., the total number of map slots plus reduce slots for the cluster is 80, as shown in Fig. 2.

Our experiments are three folds. First, we examine the influence of slot configuration to the overall performance by running jobs in all possible map/reduce slot configurations in practice (i.e., configure map slots from 1 to 7 per slave node), under an arbitrary job submission order. The experimental results are given in Fig. 2a. It can be noted that the maximum performance difference between the worst-case map/reduce slot configuration (e.g., 10/70) and the optimal one (e.g., 60/20) is huge, up to $\frac{3{,}112-624}{624} \approx 399\%$. Second, we consider the influence of job orderings on the performance by running jobs with all possible job orders, under an optimal map/reduce slot configuration (e.g., 60/20). Fig. 2b shows that the performance difference of the worst-case job submission order and the optimal one can be large up to $\frac{768-578}{578} \approx 33\%$, depending on the workload characteristic. Third, we evaluate and compare the optimal(minimum) makespan as well as its corresponding optimal map/reduce slot configuration for all possible job submission orders. Fig. 2c illustrates the optimal map/reduce slot configuration (i.e., blue and green bar) as well as its corresponding

optimal makespan (i.e., red curve) for all $4! = 24$ possible job submission orders, sorted by makespan in non-decreasing order. The results show that there are varied optimal configurations of map/reduce slots for different job submission orders. Moreover, it's worth noting that the maximum performance difference between the worst-case job order and the optimal job order each under its corresponding optimal map/reduce slot configuration, is $\frac{744-578}{578} \approx 28.7\%$.

In summary, the above motivating example poses three key challenging issues: (1). Different map/reduce slot configurations will have different performance under a given job order (e.g., Fig. 2a); (2). Even under the optimal map/reduce slot configuration, different job submission orders will result in varied performance (e.g., Fig. 2b); (3). The optimal configurations of map/reduce slots as well as its corresponding optimal makespan are different under different job submission orders (e.g., Fig. 2c).

3 Related Work

In this section, we give an overview of related work from two aspects. First, we review batch job ordering optimization work in HPC literature. Second, we summarize the MapReduce job optimization work proposed in recent years.

3.1 Job Ordering Optimization

The batch job ordering problem has been extensively studied in the high performance computing literature [25]. Minimizing the makespan has been shown to be NP-hard [25], and a number of approximation and heuristic algorithms (e.g., [14], [34]) have been proposed. In addition, there has been work on bi-criteria optimization which aims to minimize makespan and total completion time simultaneously, such as [13].

The previous works all focused on the single-stage parallelism, where each job only has a single stage. In contrast, MapReduce is an interleaved parallel and sequential computation model [23] which is related to the two-stage hybrid flow shop (2HFS) problem [17]. Minimizing the makespan for 2HFS is strongly $\mathcal{NP}$-hard when at least one stage contains multiple processors [16]. There has been a large body of approximation and heuristic algorithms (e.g., [6], [24]) proposed for 2HFS. Additionally, there has been work (e.g., [31]) targeted at the bi-criteria optimization of both makespan and total completion time.

The main difference between MapReduce and traditional 2HFS is that MapReduce jobs can run multiple map and reduce tasks concurrently in each phase, whereas 2HFS allows at most one task to be processed at a time. In this way, MapReduce is more similar to the two-stage hybrid flow shop with multiprocessor tasks (2HFSMT) [28], [29] problem, which allows a task at each stage can be processed on multiple processors simultaneously. However, there is a requirement in 2HFSMT that a task at each stage can be scheduled only when the number of processors it requires is satisfied; otherwise the task needs to wait [28]. In contrast, the number of running map/reduce tasks for a MapReduce job can be dynamically scaled up and down as idle map/reduce slots become available.

In summary, MapReduce is a new computation model that is similar to but different from other models mentioned above. The works that are most related to ours are [26], [41]. In [26], Moseley et al. present an offline 12-approximation algorithm for minimizing the *total flow time* of the jobs; this is the sum of the differences between the finishing and arrival times of all the jobs. Verma et al. [41] propose two algorithms for makespan optimization. One is a greedy algorithm job ordering method based on Johnson's Rule. Another is a heuristic algorithm called BalancedPool. They discuss and evaluate the algorithms experimentally. We follow their job ordering approach (i.e., *MK_JR* algorithm in our paper). But our main contributions go beyond it in a number of significant aspects. First, we prove a $1 + \delta$ upper bound on the approximation ratio of our *MK_JR* algorithm. Second, we give the relationship between upper-bound makespan, lower-bound makespan, and the corresponding job orders. Additionally, our *MK_TCT_JR* algorithm obtains a trade-off in the makespan and total completion time, which produces very good results. Moreover, for online workloads, we proposed a prototype named MROrder [36] to perform online job ordering optimization by incorporating *MK_JR* algorithm.

3.2 MapReduce Job Optimization

There is a large body of research work that focuses on the optimization for MapReduce jobs. One optimization policy focuses on the architectural design and optimization issues. Jiang et al. [21] proposed a set of general low-level optimizations including improving I/O speed, utilizing indexes, using fingerprinting for faster key comparisons, and block size tuning. Rasmussen et al. [33] presented an I/O-efficient MapReduce system called *Themis* that improves the performance of MapReduce by minimizing the number of I/O operations. Likewise, Sailfish [32] improves MapReduce's performance through more efficient disk I/O. It mitigates partitioning skew in MapReduce by choosing the number of reduce tasks and intermediate data partitioning dynamically at runtime, using an index constructed on intermediate data. There are also methods that reduce I/O cost in MapReduce by using indexing structures (e.g., Hadoop++ [12]), column-oriented storage (e.g., [15]). Polo et al. [30] proposed a scheduling technique and implemented a prototype called Adaptive Scheduler that can adaptively manage the workload performance with the awareness of hardware heterogeneity, distributed storage to meet user's deadline requirement. Wolf et al. [42] propose a flexible scheduling allocation scheme called FLEX, which can optimize any of a variety of standard scheduling theory metrics, such as response time, stretch, makespan. Tang et al. [35], [37] proposed a dynamic slot allocation system called DynamicMR to improve the performance for the slot-based Hadoop MRv1, by allowing map (or reduce) tasks can be run on map slots and reduce slots.

Adjusting Hadoop configuration is another optimization policy, including [7], [18], [19]. For example, Starfish [19] is a self-tuning framework that can adjust the Hadoop's configuration automatically for a MapReduce job such that the utilization of Hadoop cluster can be maximized, based on the cost-based model and sampling technique. Herodotou and Babu [18] propose a system named *Elastisizer* for

cluster-sizing optimization and MapReduce job-level parameter configurations optimization, on the cloud platform, to meet desired requirements on execution time and cost for a given workload, based on a careful mix of job profiling, estimation using black-box and white-box models and simulation. In contrast, Agarwal et al. [7] present a system *RoPE* that can re-optimize data parallel jobs by adapting execution plans based on estimates of code and data properties.

Another optimization policy is to share work and eliminate redundant data access and computation. Agrawal et al. [8] provide a method to maximize scan sharing by grouping MapReduce jobs into batches so that sequential scans of large files are shared among many simultaneous jobs as possible. MRShare [27] is a sharing framework that provides three possible work-sharing opportunities, including scan sharing, mapped outputs sharing, and Map function sharing across multiple MapReduce jobs, to avoid performing redundant work and thereby save processing time.

There is also an optimization policy of *pipelining*. MapReduce Online [10] is such a modified MapReduce system to support online aggregation for MapReduce jobs that run continuously by pipelining data within a job and between jobs.

In contrast, we improve the performance for a MapReduce workload by maximizing the cluster utilization as much as possible, through optimizing the map/reduce slot configuration and the job submission order. All these studies are complementary to our study and our approach can be incorporated into these modified MapReduce frameworks (e.g., MRShare [27], MapReduce Online [10]) for further performance improvement.

Moreover, there are a number of optimization works for MapReduce on the cloud, which primarily consider the deadline and budget, such as [20], [38], [39]. They optimize the task scheduling and resource allocation for MapReduce workloads by proposing algorithms and cost models for each metric. However, their work are atop of Hadoop system. We can combine these works and our approach to further optimize the deadline and budget for cloud computing.

4 Problem Formulation and Performance Model

In this section, we give a formal model for MapReduce and formalize its associated optimization problems.

4.1 Problem Formulation

A MapReduce *job* J_i computation consists of two phases, a *map* phase $\mathcal{M}$ and *reduce* phase $\mathcal{R}$. Each phase consists of a number of *tasks*. We write $|J_i^{\mathcal{M}}|$ and $|J_i^{\mathcal{R}}|$ for the number of tasks in J_i's map phase and reduce phase, respectively. Let $t_{i,j}^{\mathcal{M}}$ and $t_{i,j}^{\mathcal{R}}$ denote the execution time of J_i's jth map task and jth reduce task, respectively. We consider a MapReduce *workload* with a set of independent jobs $J = \{J_1, J_2, \ldots J_n\}$, for some n. These jobs can be executed in any order. The workload is executed on a MapReduce cluster under FIFO scheduling, consisting of a set of (map and reduce) *slots*, denoted as $\mathcal{S}$. Let $\mathcal{S}^{\mathcal{M}}$ and $\mathcal{S}^{\mathcal{R}}$ denote the set of map slots and reduce slots configured by MapReduce administrator (i.e., $\mathcal{S} = \mathcal{S}^{\mathcal{M}} \cup \mathcal{S}^{\mathcal{R}}$), so that the number of map slots and reduce slots are $|\mathcal{S}^{\mathcal{M}}|$ and $|\mathcal{S}^{\mathcal{R}}|$, correspondingly.

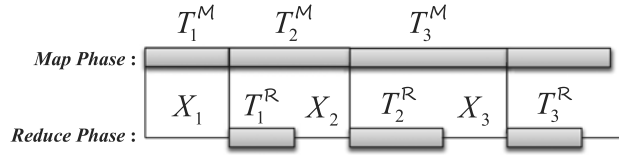


Fig. 3. MapReduce tasks execution flow for the simplified case.

Let ϕ denote the job submission order for a MapReduce workload. We focus on the *offline* situation in which all the jobs are available at time 0. Let c_i denote the completion time of J_i (i.e., the time when J_i's reduce tasks all finish). The *makespan* for the workload $J_1, \ldots, J_n$ is defined as $C_{max} = \max_{i \in [n]}\{c_i\}$. The *total completion time* for the workload is defined as $C_{tct} = \sum_{i \in [n]} c_i$.

In our work, we consider four optimization problems, defined as follows:

Problem 1. *Find an ordering ϕ to execute the jobs $J_1, \ldots, J_n$ in a MapReduce workload such that C_{max} is minimized, under a given slot configuration $(\mathcal{S}^{\mathcal{M}}, \mathcal{S}^{\mathcal{R}})$?*

Problem 2. *Find an ordering ϕ to execute the jobs $J_1, \ldots, J_n$ in a MapReduce workload that can optimize (minimize) C_{max} and C_{tct} simultaneously, under a given slot configuration $(\mathcal{S}^{\mathcal{M}}, \mathcal{S}^{\mathcal{R}})$?*

Moreover, if we are MapReduce cluster administrators, we can perform the following optimization work:

Problem 3. *Find a map/reduce slot configuration $(\mathcal{S}^{\mathcal{M}}, \mathcal{S}^{\mathcal{R}})$ and ordering ϕ to execute the jobs $J_1, \ldots, J_n$ in a MapReduce workload such that C_{max} is minimized, under a given value of total slots $\mathcal{S}$?*

Problem 4. *Find a map/reduce slot configuration $(\mathcal{S}^{\mathcal{M}}, \mathcal{S}^{\mathcal{R}})$ and ordering ϕ to execute the jobs $J_1, \ldots, J_n$ in a MapReduce workload that can optimize (minimize) C_{max} and C_{tct} simultaneously, under a given value of total slots $\mathcal{S}$?*

4.2 Performance Model for Makespan and Total Completion Time

In this section, we aim to deduce the mathematical performance model for makespan and total completion time. We start by considering a simplified case where we can give a close-form formula for makespan and total completion time. Next, we consider the general case in which it is complex and difficult to get the exact mathematical formula. Instead, we deduce an upper bound for it.

We first consider a simplified case where $|\mathcal{S}^{\mathcal{M}}| = 1$ and $|\mathcal{S}^{\mathcal{R}}| = 1$. It turns out to be a perfect two-machine flow-shop problem [41]. Fig. 3 gives an example of an execution for this case. For each job J_i, let $T_i^{\mathcal{M}}$ be the total processing time of map tasks and $T_i^{\mathcal{R}}$ be the total processing time for the reduce tasks. Let X_i be the idle period of time for reduce machines before the reduce tasks of job J_i start running. Then we have $T_i^{\mathcal{M}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|}$ and $T_i^{\mathcal{R}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|}$. Based on the Johnson's work [22], the makespan and total completion time for the simplified case can therefore be calculated as follows:

$$X_k = \max\left\{\sum_{i=1}^{k} T_i^{\mathcal{M}} - \sum_{i=1}^{k-1}(T_i^{\mathcal{R}} + X_i), 0\right\}. \quad (1)$$

$$\sum_{i=1}^{n} X_i = \max_{1\le k\le n}\left\{\sum_{i=1}^{k} T_i^{\mathcal{M}} - \sum_{i=1}^{k-1} T_i^{\mathcal{R}}\right\}. \quad (2)$$

$$C_{max} = \sum_{i=1}^{n}(X_i + T_i^{\mathcal{R}}) = \max_{1\le k\le n}\left\{\sum_{i=1}^{k} T_i^{\mathcal{M}} + \sum_{i=k}^{n} T_i^{\mathcal{R}}\right\}. \quad (3)$$

$$C_{tct} = \sum_{u=1}^{n}\sum_{i=1}^{u}(X_i + T_i^{\mathcal{R}}) = \sum_{u=1}^{n}\max_{1\le k\le u}\left\{\sum_{i=1}^{k} T_i^{\mathcal{M}} + \sum_{i=k}^{u} T_i^{\mathcal{R}}\right\}. \quad (4)$$

According to Johnson's work [22], there is a useful property about the k which maximizes the quantity above for makespan as stated by the following lemma below,

Lemma 1. *For C_{max} in Formula (3), (1). if $C_{max} = \sum_{i=1}^{k_0} T_i^{\mathcal{M}} + \sum_{i=k_0}^{n} T_i^{\mathcal{R}}$, $(1 \le k_0 \le n)$, there must be $X_{k'} = 0$ for all $k_0 < k' \le n$; (2). Conversely, if $X_{k_0} > 0$ and $X_{k'} = 0$ for all $k_0 < k' \le n$, there must be $C_{max} = \sum_{i=1}^{k_0} T_i^{\mathcal{M}} + \sum_{i=k_0}^{n} T_i^{\mathcal{R}}$, $(1 \le k_0 \le n)$.*

The detailed proof of Lemma 1 is given in Appendix A of the supplemental file, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TSC.2015.2426186. It shows us an important implication for *makespan* estimation. When it comes to a Hadoop environment for a set of ordered n jobs, k_0 is an indicator for the k_0^{th} job in the order from which all reduce slots will be fully utilized till the end of batch computation. We use it later for the proof of Lemma 2.

Next, we consider the general case where the number of map and reduce slots are arbitrary, as illustrated in Fig. 4. In this case, it is difficult to give a closed-form formula for the makespan and total completion time. Instead, we can compute the makespan and total completion time using a simple program we call *MREstimator*, which simulates the execution of a set of jobs under an ordering. Given a job ordering, *MREstimator* executes jobs in the way described in Section 4.1 to derive the makespan and total completion time. We use *MREstimator* to determine the empirical performance of algorithms *MK_JR* and *MK_TCT_JR* in Section 7. In addition to this evaluation, we need an analytical formula for the makespan in order to derive the approximation ratio of *MK_JR* in Lemma 3. As giving an exact formula for general workloads is difficult, we instead give the following upper bound on the makespan of a certain job ordering.

Lemma 2. *Given a job order ϕ, there is a upper bound makespan denoted by $\widehat{C}_{max}$ (i.e. $C_{max} \le \widehat{C}_{max}$) for the generalized case as follows:*

$$\widehat{C}_{max} = \max_{1\le k\le n}\left\{\sum_{i=1}^{k} T_i^{\mathcal{M}} + \max_{1\le i\le k}\{\widehat{t}_i^{\mathcal{M}}\} + \sum_{i=k}^{n} T_i^{\mathcal{R}} + \max_{1\le i\le n}\{\widehat{t}_i^{\mathcal{R}}\}\right\}. \quad (5)$$

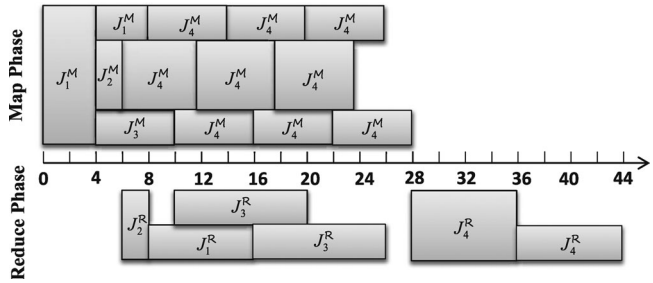


Fig. 4. MapReduce tasks execution flow for an example of four jobs under the generalized case, where $|\mathcal{S}^{\mathcal{M}}| = 8$, $|\mathcal{S}^{\mathcal{R}}| = 4$. The job submission order is $J_1 \to J_2 \to J_3 \to J_4$.

where

$$T_i^{\mathcal{M}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|},\ T_i^{\mathcal{R}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{R}}|},\ \widehat{t}_i^{\mathcal{M}} = \max_{1\le j\le |J_i^{\mathcal{M}}|} \widehat{t}_{i,j}^{\mathcal{M}}$$

and $\widehat{t}_i^{\mathcal{R}} = \max_{1\le j\le |J_i^{\mathcal{R}}|} \widehat{t}_{i,j}^{\mathcal{R}}$.

The detailed proof of Lemma 2 is given in Appendix B of the supplemental file, available online.

5 Job Ordering Optimization for MapReduce Workload

This section attempts to address Problem 1 and Problem 2. We first focus on makespan optimization. We describe the *MK_JR* algorithm that produces the optimized job order and also prove its approximation ratio. We also describe the job order which gives the *worst*, i.e., longest makespan, which is used for derivation of the upper bound makespan of a workload. Next, we describe the *MK_TCT_JR* algorithm, which optimizes both makespan and total completion time. Finally, Section 5.3 shows that the orderings produced by *MK_JR* and *MK_TCT_JR* are stable, even when MapReduce servers fail.

5.1 Makespan Optimization

Recall the simplified case described in the previous section, where the number of map and reduce tasks of all the jobs were divisible by the number of map and reduce slots. The optimal job order for the simplified case can be obtained by using *Johnson's Rule* [22], which is an efficient $O(n \log n)$ job ordering algorithm for the minimum makespan C_{max}^{opt} for the two-stage flow shop with one processor per stage. Johnson's rule works as follows. Divide the jobs set J into two disjoint sub-sets J_A and J_B. Set J_A consists of those jobs J_i for which $T_i^{\mathcal{M}} < T_i^{\mathcal{R}}$. Set J_B contains the remaining jobs (i.e. $J \setminus J_A$). Sequence jobs in J_A in non-decreasing order of $T_i^{\mathcal{M}}$ and those in J_B in non-increasing order of $T_i^{\mathcal{R}}$. The optimal job order is obtained by appending the sorted set J_B to the end of sorted set J_A. When the number of tasks is not divisible by the number of slots, the makespan minimization problem becomes NP-hard. Verma et al. [41] first noted it and proposed an algorithm based on Johnson's rule. as we re-format in the following algorithm *MK_JR*. Its time complexity is $O(n \log n)$. To well study the makespan result on it, in the rest part, we give a theoretical analysis based on our previous performance model in Section 4.2.

Theorem 1. *MK_JR is an $(1+\delta)$-approximation algorithm for makespan optimization in the generalized case, where* $\delta = \frac{\max_{1\le i\le n}\{\widehat{t_i^{\mathcal{M}}}\}+\max_{1\le i\le n}\{\widehat{t_i^{\mathcal{R}}}\}}{\max_{1\le k\le n}\{\sum_{i=1}^{k} T_i^{\mathcal{M}}+\sum_{i=k}^{n} T_i^{\mathcal{R}}\}}, (0 \le \delta \le 1)$.

To prove Theorem 1, we need to compute an upper bound and a lower bound on the makespan of all job orderings.

Lemma 3. *Let $\check{\phi}^*$ denote the job order produced by MK_JR. Let $\widehat{\phi}^*$ be the reversing order of $\check{\phi}^*$. Then there is a lower bound makespan denoted by $\check{C}^*_{max}$, as well as a upper bound makespan denoted by $\widehat{C}^*_{max}$, for all job orders Φ. Particularly, $\check{C}^*_{max}$ is estimated with regard to $\check{\phi}^*$ by using Formula (3). $\widehat{C}^*_{max}$ is estimated with regard to $\widehat{\phi}^*$ with the Formula:*

$$\widehat{C}^*_{max} = \max_{1\le k\le n}\left\{\sum_{i=1}^{k} T_i^{\mathcal{M}} + \max_{1\le i\le n}\{\widehat{t_i^{\mathcal{M}}}\} + \sum_{i=k}^{n} T_i^{\mathcal{R}} + \max_{1\le i\le n}\{\widehat{t_i^{\mathcal{R}}}\}\right\}. \tag{6}$$

Algorithm 1. Greedy Algorithm Based on *Johnson's Rule* (**MK_JR**)

Input:
J: the MapReduce workload.
$|\mathcal{S}^{\mathcal{M}}|$: the given number of map slots.
$|\mathcal{S}^{\mathcal{R}}|$: the given number of reduce slots.

Output:
ϕ : the optimized job submission order.

1: For each job J_i, we first estimate its map-phase processing time $T_i^{\mathcal{M}}$ and reduce-phase processing time $T_i^{\mathcal{R}}$ by using the following formula:

$$(T_i^{\mathcal{M}}, T_i^{\mathcal{R}}) = \left(\frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|}, \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|} \cdot t_i^{\mathcal{R}}\right).$$

2. We order jobs in J based on the following principles:
   a). Partition jobs set J into two disjoint sub-sets J_A and J_B:

$$J_A = \{J_i | (J_i \in J) \wedge (T_i^{\mathcal{M}} \le T_i^{\mathcal{R}})\},$$
$$J_B = \{J_i | (J_i \in J) \wedge (T_i^{\mathcal{M}} > T_i^{\mathcal{R}})\}.$$

   b). Order all jobs in J_A from left to right by non-decreasing $T_i^{\mathcal{M}}$. Order all jobs in J_B from left to right by non-increasing $T_i^{\mathcal{R}}$.
   c). Make an ordered jobs set $J^{'}$ by joining all jobs in J_A first and then J_B in order, i.e., $\phi_1 : J^{'} = \{(J_A), (J_B)\}$.

To prove Lemma 3, we need to find a worst-case job order for a batch of jobs in the simplified case of MapReduce such that its makespan, denoted as C^{wst}_{max}, is maximized. We find the following interesting relationship optimal job order and the least optimal job order, for the simplified case where the number of (map or reduce) tasks divides the number of (map or reduce) slots.

Theorem 2. *Suppose ϕ is the optimal job order whose makespan is C^{opt}_{max}, based on Johnson's Rule for the two-stage flow shop with one processor per stage. The worst-case job order ϕ^*, whose makespan is C^{wst}_{max}, can be obtained by simply reversing ϕ.*

The proof of Theorem 2 is given in Appendix C of the supplemental file, available online. We now can prove Lemma 3 based on Theorem 2 and Lemma 2. The detailed proof of Lemma 3 is given in Appendix D of the supplemental file, available online.

Finally, the main Theorem 1 can be proved based on Lemma 2 and Lemma 3, as given in Appendix E of the supplemental file, available online.

Note that in general the execution time for a single map/reduce task is much smaller compared with the total execution time (i.e., δ is very small). It means that *MK_JR* has a good approximation in the generalized case.

5.2 Bi-Criteria Optimization of Makespan and Total Completion Time

Makespan and *total completion time* are two key performance metrics. Generally, *makespan* refers to the maximum completion time for a batch of jobs. It considers the computation time of jobs and is often used to measure the performance and utilization efficiency of a system. In contrast, *total completion time* is the sum of completion time of all jobs. It is a generalized makespan with queuing time (i.e., waiting time) included. It can be used to measure the satisfaction to the system from a single job's perspective. So far, we focus only on the optimization of makespan. Note that the total completion time that can be poor subject to obtaining optimal makespan, as illustrated in Fig. 1a. Therefore, there is a need for bi-criteria optimization on both makespan and total completion time. Intuitively, the makespan is affected primarily by the positions of large-size jobs. In contrast, the total completion time is mainly influenced by the positions of small-size jobs. The algorithm shortest processing time first (SPTF) is optimal for the total completion time on a single machine where there is one task per job and no precedence constrains [5]. However, *MK_JR* is not aware of varying job sizes. Indeed, the job order produced by *MK_JR* can have adverse effect on the total completion time if we follow *Johnson's Rule* strictly in some scenarios. For example, there can be a job J_i whose processing time (e.g., $T_i^{\mathcal{M}} + T_i^{\mathcal{R}}$) is very small but $T_i^{\mathcal{M}} > T_i^{\mathcal{R}}$. We should schedule J_i early if we want to minimize the total completion time, whereas *MK_JR* might put it in the middle or later part of the order list according to *Johnson's Rule*. We therefore design a new greedy algorithm *MK_TCT_JR* on top of *MK_JR* by combining *SPTF* and *Johnson's Rule*. The time complexity of *MK_TCT_JR* is $O(n \log n)$.

In *MK_TCT_JR*, we first divide job set J into two subsets, $J^{'}_A$ and $J^{'}_B$. Let $J^{'}_A$ contain small-size jobs and $J^{'}_B$ contain large-size jobs. We schedule jobs in $J^{'}_A$ first and then $J^{'}_B$. Within each set, we use *MK_JR* to minimize its makespan. We estimate the processing time for each job by adding its map-phase running time and reduce-phase running time, given the whole map/reduce slots of the Hadoop cluster. Particularly, our classification of small-/large-size jobs is based on the geometric mean of processing time of all jobs, considering that unlike the arithmetic mean that favors large-size jobs, geometric mean has a good *unbiased* property for all jobs [9]. Note that moving small-size jobs forward benefits the total completion time but will hurt the makespan. Geometric mean is a tradeoff choice based on

the reason that we want to improve the performance of total completion time while not hurt the makespan seriously.

5.3 Analysis of Job Ordering Algorithms

In a MapReduce cluster, auto-scaling allows us to add or remove some slave nodes from the cluster during the computation dynamically. It has been supported by the current implementation of Hadoop. Moreover, the failure of a machine is unavoidable, which can cause a node removed from the MapReduce cluster dynamically. Therefore, an issue arises about its influence on the optimized job orders produced by *MK_JR* and *MK_TCT_JR*.

For job ordering algorithms, we observe an interesting finding regarding this issue as follows. Given a homogeneous environment where the Hadoop configurations of slave nodes are identical, there is an important feature for the optimized job orders produced by *MK_JR* and *MK_TCT_JR* as follows:

Theorem 3. *Given a homogeneous environment where the Hadoop configurations of slave nodes are identical, the job orders ϕ_1 produced by MK_JR and ϕ_2 produced by MK_TCT_JR for a batch of jobs are independent of the number of slave nodes, but rather depend on the number of map/reduce slots configured within a slave node.*

The proof of Theorem 3 is shown in Appendix F of the supplemental file, available online. It gives us an important implication that the *optimized* job orders produced by *MK_JR* and *MK_TCT_JR* are *stable* (unchanged) when we dynamically add or remove slave nodes to the Hadoop cluster. Particularly, it's worth mentioning that Theorem 3 does not mean the *optimal* job orders are *unchanged* under varied number of nodes.

Algorithm 2. Greedy algorithm based on *Shortest Processing Time First* and *Johnson's Rule* (***MK_TCT_JR***)

Input:
J: the MapReduce workload.
$|\mathcal{S}^{\mathcal{M}}|$: the given number of map slots.
$|\mathcal{S}^{\mathcal{R}}|$: the given number of reduce slots.
Output:
ϕ : the optimized job submission order.

1. For each job J_i, we first compute its processing time T_i by using the formula below:

$$T_i = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|} + \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|}.$$

2. Let $T = \left(\prod_{1 \le i \le n} T_i\right)^{\frac{1}{n}}$. We divide jobs set J into two disjoint sub-sets $J_A^{'}$ and $J_B^{'}$:

$$J_A^{'} = \{J_i | (J_i \in J) \wedge (T_i \le T)\}, \quad J_B^{'} = \{J_i | (J_i \in J) \wedge (T_i > T)\}$$

3. Order all jobs in $J_A^{'}$ and $J_B^{'}$ using *MK_JR* respectively.
4. Make a ordered jobs set $J^{'}$ by joining all jobs in the ordered set $J_A^{'}$ first and then the ordered set $J_B^{'}$, i.e., $\phi_2 : J^{'} = \{\{J_A^{'}\}, \{J_B^{'}\}\}$.

For example, Let us consider a Hadoop cluster with five nodes, each configured with two map and two reduce slots. Let J_1 be defined as follows: Map stage duration is 9 and requires 10 map slots. Reduce stage duration is 10 and requires one reduce slot. Let J_2 be defined as follows: Map stage duration is 11 and requires eight map slots and reduce stage duration is 15 and requires one reduce slot. In this case, the optimal job scheduling order is $J_1 \rightarrow J_2$, with the makespan of 35. The optimized job order produced by *MK_JR* is also $J_1 \rightarrow J_2$.

Now, if one node fails, then there are only four nodes left with eight map and eight reduce tasks available in the cluster. In this case, the *optimized* job order generated by *MK_JR* keeps the same, i.e., $J_1 \rightarrow J_2$, with the makespan of 44. However, the optimal job scheduling is $J_2 \rightarrow J_1$ with the makespan of 39.

6 Slot Configuration Optimization for MapReduce Workload

In this section, we attempt to solve Problem 3 and Problem 4. We first propose some map/reduce slot configuration algorithms to optimize makespan. Then, the bi-criteria algorithm *MK_TCT_SF_JR* is described to optimize the makespan and total completion time together. Finally, we give a discussion and analysis for slot configuration optimization algorithms.

6.1 Makespan Optimization

Given a MapReduce workload and the total number of slots, an intuitive method is to search and compare all combinations of job submission orders and map/reduce slot configurations exhaustively, as shown in Algorithm 3. It can produce the optimal execution plan for makespan optimization. However, its time complexity is $O(n! \times |S| \times n)$, which is exponential and thus unacceptable for large-size number of jobs in practice.

Algorithm 3. Brute-force Search algorithm for the optimal map/reduce slot configurations and job submission orders. (***EX_MK_SF_JR***)

Input:
J: the MapReduce workload.
$|\mathcal{S}|$: the total number of slots.
Output:
ϕ: the optimized job submission order.
$|\mathcal{S}^{\mathcal{M}}|$: the optimized number of map slots.
$|\mathcal{S}^{\mathcal{R}}|$: the optimized number of reduce slots.
$Mini\_Makespan$: the minimized *makespan*.

1: $Mini\_Makespan \leftarrow \infty$, $\phi \leftarrow null$.
2: **for each** $\phi_0 \in \Phi$ **do** //Φ denotes the set of all job submission orders.
3: **for** $|\mathcal{S}_0^{\mathcal{M}}|$ from 1 to $|\mathcal{S}| - 1$ **do**
4: $|\mathcal{S}_0^{\mathcal{R}}| \leftarrow |\mathcal{S}| - |\mathcal{S}_0^{\mathcal{M}}|$.
5: $Makespan \leftarrow MREstimator\ (J, \phi_0, |\mathcal{S}_0^{\mathcal{M}}|, |\mathcal{S}_0^{\mathcal{R}}|)$.
6: **if** $Mini\_Makespan > Makespan$ **then**
7: $Mini\_Makespan \leftarrow Makespan$.
8: $(|\mathcal{S}^{\mathcal{M}}|, |\mathcal{S}^{\mathcal{R}}|) \leftarrow (|\mathcal{S}_0^{\mathcal{M}}|, |\mathcal{S}_0^{\mathcal{R}}|)$.
9: $\phi \leftarrow \phi_0$.
10: **end if**
11: **end for**
12: **end for**
13: **return** (ϕ, $|\mathcal{S}^{\mathcal{M}}|$, $|\mathcal{S}^{\mathcal{R}}|$, $Mini\_Makespan$).

It is worth noting in Algorithm EX_MK_SF_JR that, the exhaustive search of all possible job submission orders has time complexity of $O(n!)$, which is a serious bottleneck on performance. To alleviate the performance bottleneck, instead, we can include efficient job ordering optimization algorithms, such as the previously proposed Algorithm MK_JR, as shown in Algorithm 4. It is an $O(nlogn)$ $(1+\delta)$-approximation algorithm for makespan under a given map/reduce slot configuration, where $\delta < 1$ is the ratio of sum of the maximum map and reduce task size, to the sum of all the task sizes.

Algorithm 4. Search algorithm for optimized slot configuration and job submission order. (***MK_SF_JR***)

Input:
J: the MapReduce workload.
$|\mathcal{S}|$: the total number of slots.
Output:
ϕ: the optimized job submission order.
$|\mathcal{S}^{\mathcal{M}}|$: the optimized number of map slots.
$|\mathcal{S}^{\mathcal{R}}|$: the optimized number of reduce slots.
$Mini\_Makespan$: the minimized *makespan*.

1: $Mini\_Makespan \leftarrow \infty, \phi \leftarrow null$.
2: **for** $|\mathcal{S}_0^{\mathcal{M}}|$ from 1 to $|\mathcal{S}|-1$ **do**
3: $|\mathcal{S}_0^{\mathcal{R}}| \leftarrow |\mathcal{S}| - |\mathcal{S}_0^{\mathcal{M}}|$.
4: $\phi_0 \leftarrow$ MK_JR(J, $|\mathcal{S}_0^{\mathcal{M}}|$, $|\mathcal{S}_0^{\mathcal{R}}|$).
5: $Makespan \leftarrow MREstimator\ (J, \phi_0, |\mathcal{S}_0^{\mathcal{M}}|, |\mathcal{S}_0^{\mathcal{R}}|)$.
6: **if** $Mini\_Makespan > Makespan$ **then**
7: $Mini\_Makespan \leftarrow Makespan$.
8: $(|\mathcal{S}^{\mathcal{M}}|, |\mathcal{S}^{\mathcal{R}}|) \leftarrow (|\mathcal{S}_0^{\mathcal{M}}|, |\mathcal{S}_0^{\mathcal{R}}|)$.
9: $\phi \leftarrow \phi_0$.
10: **end if**
11: **end for**
12: **return** (ϕ, $|\mathcal{S}^{\mathcal{M}}|$, $|\mathcal{S}^{\mathcal{R}}|$, $Mini\_Makespan$).

Theorem 4. *Algorithm MK_SF_JR is an* $(1+\delta)$*-approximation algorithm for makespan optimization, where* $\delta = \frac{\max_{1\le i\le n}\{\widehat{t_i^{\mathcal{M}}}\}+\max_{1\le i\le n}\{\widehat{t_i^{\mathcal{R}}}\}}{\max_{1\le k\le n}\{\sum_{i=1}^{k} T_i^{\mathcal{M}}+\sum_{i=k}^{n} T_i^{\mathcal{R}}\}}, (0 \le \delta \le 1)$, *under the optimal map/reduce slot configuration.*

The proof of Theorem 4 is simple and we omit it here.

6.2 Bi-Criteria Optimization of Makespan and Total Completion Time

Recall in Section 5.2, we have shown that the total completion time can be poor subject to obtaining the optimal makespan. Likewise, there is also a need to optimize them together when we do the slot configuration optimization. As illustrated in Algorithm 5, it is a bi-criteria optimization algorithm for makespan and total completion time with regard to slot configuration optimization. It is a search algorithm that incorporates the bi-criteria job ordering algorithm *MK_TCT_JR*.

6.3 Discussion and Analysis of Slot Configuration Algorithms

It is worth noting that, for the above slot allocation algorithms (i.e., Algorithm EX_MK_SF_JR, Algorithm MK_SF_JR, and Algorithm MK_TCT_SF_JR), we obtain the optimized map/reduce slot configuration by enumerating and validating all of their possible combinations from 1 to $|\mathcal{S}|-1$. However, there might be an efficiency problem for these search algorithms when the total number of slots $|\mathcal{S}|$ is very large (e.g., $|\mathcal{S}| = 1{,}000{,}000$). To address such an issue, interestingly, we find a very important '*proportional configuration*' property that can overcome the efficiency problem for very large value of $|\mathcal{S}|$ as follows:

Algorithm 5. Search algorithm for optimized slot configuration and job submission order. (***MK_TCT_SF_JR***)

Input:
J: the MapReduce workload.
$|\mathcal{S}|$: the total number of slots.
Output:
ϕ: the optimized job submission order.
$|\mathcal{S}^{\mathcal{M}}|$: the optimized number of map slots.
$|\mathcal{S}^{\mathcal{R}}|$: the optimized number of reduce slots.
$Mini\_Makespan$: the optimized *makespan*.
$Mini\_TCT$: the optimized *total completion time*.

1: $Mini\_Makespan \leftarrow \infty, \phi \leftarrow null$.
2: **for** $|\mathcal{S}_0^{\mathcal{M}}|$ from 1 to $|\mathcal{S}|-1$ **do**
3: $|\mathcal{S}_0^{\mathcal{R}}| \leftarrow |\mathcal{S}| - |\mathcal{S}_0^{\mathcal{M}}|$.
4: $\phi_0 \leftarrow$ MK_TCT_JR(J, $|\mathcal{S}_0^{\mathcal{M}}|$, $|\mathcal{S}_0^{\mathcal{R}}|$).
5: $(Makespan, TCT) \leftarrow MREstimator\ (J, \phi_0, |\mathcal{S}_0^{\mathcal{M}}|, |\mathcal{S}_0^{\mathcal{R}}|)$.
6: **if** $Mini\_Makespan > Makespan$ **then**
7: $Mini\_Makespan \leftarrow Makespan$.
8: $Mini\_TCT \leftarrow TCT$.
9: $(|\mathcal{S}^{\mathcal{M}}|, |\mathcal{S}^{\mathcal{R}}|) \leftarrow (|\mathcal{S}_0^{\mathcal{M}}|, |\mathcal{S}_0^{\mathcal{R}}|)$.
10: $\phi \leftarrow \phi_0$.
11: **end if**
12: **end for**
13: **return** (ϕ, $|\mathcal{S}^{\mathcal{M}}|$, $|\mathcal{S}^{\mathcal{R}}|$, $Mini\_Makespan$, $Mini\_TCT$).

Proportional Configuration Property (PCP). Instead of searching the space for all combinations of map/reduce slots when the total number of slots $|\mathcal{S}|$ is very large, the optimal result $(|\mathcal{S}^{\mathcal{M}}|, |\mathcal{S}^{\mathcal{R}}|)$ of its map/reduce slot configuration can be estimated based on the optimal result $(|\mathcal{S}_0^{\mathcal{M}}|, |\mathcal{S}_0^{\mathcal{R}}|)$ of a small-size total number of slots $|\mathcal{S}_0|(|\mathcal{S}_0| < |\mathcal{S}|)$, with $(|\mathcal{S}^{\mathcal{M}}| = \frac{|\mathcal{S}|\cdot\mathcal{S}_0^{\mathcal{M}}}{|\mathcal{S}_0|}, |\mathcal{S}^{\mathcal{R}}| = \frac{|\mathcal{S}|\cdot\mathcal{S}_0^{\mathcal{R}}}{|\mathcal{S}_0|})$. Moreover, the optimized job submission order for the case of large number of slots $|\mathcal{S}|$ is the same as that of small-size number of slots $|\mathcal{S}_0|$.

Given $|S| = 1{,}000{,}000$, for example, it might be time-consuming to compute its optimized map/reduce slot configuration using the above algorithms directly. Instead, we can compute the optimized slot configuration directly with the above algorithms at a small-scale (e.g., $|S_0| = 1{,}000$) with the above algorithms first. Then in terms of *PCP*, we can estimate the optimized map/reduce slot configuration based on the result of $|S_0|$. Particularly, *PCP* is based Theorem 3 and Lemma 4 as follows:

Lemma 4. *Let* ρ *be the ratio of map slots to reduce slots, i.e.,* $\rho = \frac{|\mathcal{S}^{\mathcal{M}}|}{|\mathcal{S}^{\mathcal{R}}|}$. *The optimal configuration of* ρ *in the simplified case for makespan* C_{max} *and total completion time* C_{tct} *are all independent of the total number of slots* $|\mathcal{S}|(|\mathcal{S}| = |\mathcal{S}^{\mathcal{M}}| + |\mathcal{S}^{\mathcal{R}}|)$, *but rather depends on the MapReduce workload as well as its job submission order* ϕ.

TABLE 1
The Job Information for Purdue MapReduce Benchmarks

Benchmark	# of map tasks	# of reduce tasks	Execution time for map tasks (sec)		Execution time for reduce tasks (sec)	
			Average Time (sec)	Standard deviation	Average Time (sec)	Standard deviation
Wordcount	160	100	22	2.09	11	0.67
Sort	320	200	9	3.07	24	2.63
Grep	480	120	9	0.8	11	0.47
Inverted-Index	640	100	32	3.0	23	2.6
Classification	160	120	6	0.55	13	1.03
Histogram-Movies	160	150	6	0.54	13	1.3
Histogram-Ratings	160	100	18	1.18	15	1.76
Sequence-Count	320	150	38	4.5	21	2.4
Tera-Sort	160	100	10	2.7	26	3.8

The proof of Lemma 4 is given in Appendix G of the supplemental material, available online. It gives us an important insight that, the configuration ratio ρ of map slots to reduce slots for makespan and total completion time for a MapReduce workload under a given job submission order in the simplified case, are independent of the total number of slots $|\mathcal{S}|$. We then have that *PCP* holds for Algorithm EX_MK_SF_JR. Because the optimal value of ρ keeps unchanged for varied sizes of the total number of slots, which instead depends on the optimal job order for Algorithm EX_MK_SF_JR. Moreover, Theorem 3 also indicates that, the optimized job orders produced by job ordering algorithms *MK_JR* and *MK_TCT_JR* keep unchanged when varying the total number of slots, given that the configuration ratio ρ of map slots to reduce slots is fixed. All of these indicate that the optimal configuration ratio ρ and optimized job orders produced by Algorithm MK_SF_JR and Algorithm MK_TCT_SF_JR for makespan and total completion time are all independent of the total number of slots $|\mathcal{S}|$. In other words, optimal configuration ratio ρ and optimized job orders remain the same under varied values of $|\mathcal{S}|$ for the makespan and total completion time in the simplified case. Hence we have that *PCP* also holds for Algorithm MK_SF_JR and Algorithm MK_TCT_SF_JR. In summary, we can conclude that *PCP* can be used to address the efficiency problem in the case that $|\mathcal{S}|$ is very large for Algorithms EX_MK_SF_JR, MK_SF_JR, MK_TCT_SF_JR.

TABLE 2
The Batch Jobs Information

Job ID	Benchmark	Job ID	Benchmark
J_1	WordCount	J_{16}	Sort
J_2	Sort	J_{17}	HistogramRatings
J_3	Grep	J_{18}	Grep
J_4	InvertedIndex	J_{19}	InvertedIndex
J_5	Classification	J_{20}	HistogramRatings
J_6	HistogramMovies	J_{21}	Classification
J_7	HistogramRatings	J_{22}	TeraSort
J_8	SequenceCount	J_{23}	Grep
J_9	TeraSort	J_{24}	InvertedIndex
J_{10}	Classification	J_{25}	SequenceCount
J_{11}	WordCount	J_{26}	Sort
J_{12}	InvertedIndex	J_{27}	WordCount
J_{13}	SequenceCount	J_{28}	HistogramRatings
J_{14}	TeraSort	J_{29}	Classification
J_{15}	HistogramMovies	J_{30}	SequenceCount

The detailed information for each job is given by Table 1. Our testbed experiments consider three workloads: 10 jobs ($J_1 \sim J_{10}$), 20 jobs ($J_1 \sim J_{20}$), 30 jobs ($J_1 \sim J_{30}$).

7 Evaluation

In this section, we evaluate our proposed algorithms using two different kinds of workloads, i.e., testbed workload and synthetic Facebook workload. Our evaluation methodology is that, we first ran experiments in Amazon's elastic compute cloud (EC2) [1] with a testbed workload consisting of multiple jobs, as listed in Tables 1 and 2. Our EC2 Hadoop cluster consists of 20 nodes each belonging to an "Extra Large" VM. We configure one node as master and namenode, and the other 19 nodes as slaves and datanodes. Each "Extra Large" instance has four virtual cores with 2 EC2 compute units each, 15 GB RAM and four 420 GB hard disks [1]. Second, we consider a real workload case by generating a synthetic Facebook workload and develop a program called *MREstimator* to compute the makespan and total completion time. We perform experiments with the Facebook workload in Section 7.2. Third, we make a sensitive analysis for the impact of the inaccurate estimation of task execution time on proposed algorithms in Appendix K of the supplemental material, available online, showing that the impact is minor and it is fine to take the average execution time for map/reduce tasks as input. Moreover, we evaluate the tightness of proposed lower bound makespan as well as upper bound makespan with Facebook workloads experimentally in Appendix H of the supplemental material, available online. We show the accuracy and efficiency of *PCP* proposed in Section 6.3 in Appendix I of the supplemental material, available online. Finally, we validate the accuracy of *MREstimator* with the testbed workload through real experiments in Appendix J of the supplemental material, available online.

7.1 Experiment Result with Testbed Workload

To well reflect practical workloads, we generate our testbed workloads by choosing nine benchmarks arbitrarily from Purdue MapReduce Benchmarks Suite[1] and using their provided datasets. The detailed benchmarks are described as follows.

1. http://web.ics.purdue.edu/ fahmad/benchmarks.htm

(a) *Makespan speedup* under different number of jobs

(b) *Total completion time speedup* under different number of jobs

Fig. 5. Normalized experimental results for testbed workloads under different job submission orders in a Hadoop cluster with $|\mathcal{S}^{\mathcal{M}}| = 57$ and $|\mathcal{S}^{\mathcal{R}}| = 19$.

- *WordCount*. Computes the occurrence frequency of each word in a document.
- *Sort*. Sorts the data in the input files in a dictionary order.
- *Grep*. Finds the matches of a regex in the input files.
- *InvertedIndex*. Takes a list of documents as input and generates word-to-document indexing.
- *Classfication*. Classifies the input into one of k predetermined clusters.
- *HistogramMovies*. Generates a histogram of input data and is a generic tool used in many data analyses.
- *HistogramRatings*. Generates a histogram of the ratings as opposed to that of the movies based on their average ratings.
- *SequenceCount*. Generates a count of all unique sets of three consecutive words per document in the input data.
- *TeraSort*. Sorts 100-byte <key,value> tuples on the keys where key is a 10-byte field and the rest of the bytes as value (payload).

We evaluate our algorithms with the average execution time for map and reduce tasks. Particularly, we validate that it is suitable for using average execution time in our algorithms by showing that the impact of varying task execution time is minor in Appendix K of the supplemental material experimentally, available online. Table 1 lists the job information for our testbed workloads. It is a mix of nine benchmarks together with different sizes of input data. For each job J_i, we estimate its average task execution time $\tilde{t}_i^{\mathcal{M}}$ and $\tilde{t}_i^{\mathcal{R}}$ as shown in Table 1. Three different sizes of testbed workloads are generated, i.e., 10 jobs ($J_1 \sim J_{10}$), 20 jobs ($J_1 \sim J_{20}$), 30 jobs ($J_1 \sim J_{30}$), with these 9 benchmarks as described in Table 2.

7.1.1 Job Ordering Optimization Algorithms

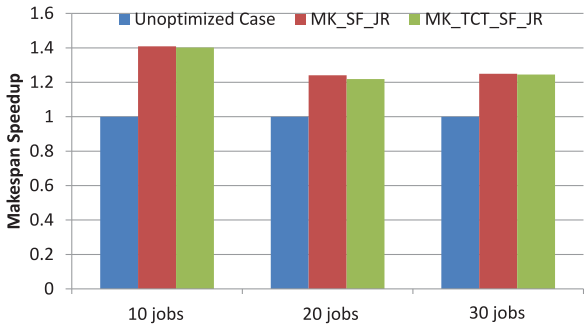
Let's begin with the evaluation of job ordering optimization algorithms *MK_JR* and *MK_TCT_JR* first. Fig. 5 presents the normalized performance results for testbed workloads under three different job orders, i.e., an unoptimized job order based on Theorem 2, the job order based on *MK_JR* and the job order based on *MK_TCT_JR*, in a Hadoop cluster, where we configure three map and one reduce slots per slave node. Therefore, we have $|\mathcal{S}^{\mathcal{M}}| = 57$ and $|\mathcal{S}^{\mathcal{R}}| = 19$. For makespan (or total completion time), we normalize it by using *makespan speedup* (or total completion time speedup), defined as the ratio of makespan (or total completion time) from the unoptimized case to that from the designated job order.

Therefore, the larger speedup indicates the better performance it is for the designated job order. As shown in Fig. 5a, the (optimized) job order based on *MK_JR* has a significant makespan improvement in comparison with an unoptimized job order. For example, there is about 24 percent performance improvement of makespan for the testbed workload with 20 jobs.

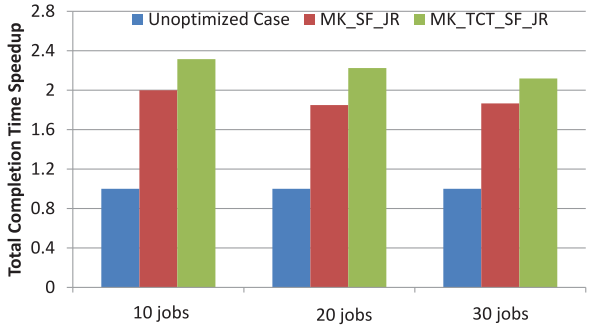
Moreover, there is a slight drop in *makespan speedup* for *MK_TCT_JR* in comparison to *MK_JR*, sacrificing a bit performance improvement in makespan for a good total completion time. It can be noted in Fig. 5b that *MK_TCT_JR* has a good *total completion time speedup*. Note that the total completion time is dominated by the position of small jobs. In our testbed workloads, since most jobs are large-size (relative to the number of map/reduce slots in our Hadoop cluster), *MK_JR* does not show much negative impact in total completion time. But for Facebook workloads which have lots of small-size jobs, we will show in Section 7.2.1 (i.e., Fig. 7b) that there is a very seriously negative impact in total completion time for *MK_JR*.

7.1.2 Slot Configuration Optimization Algorithms

In this section, let's come to evaluate map/reduce slot configuration optimization algorithms, namely, Algorithm MK_SF_JR and Algorithm MK_TCT_SF_JR. Fig. 6 illustrates experimental results for testbed workloads under various slot configuration optimization algorithms. Particularly, we take the Hadoop default configuration which sets each slave node with two map slots and two reduce slots as the unoptimized case. Then for 19 slave nodes, it holds $|\mathcal{S}^{\mathcal{M}}| = 38$ and $|\mathcal{S}^{\mathcal{R}}| = 38$. Fig. 6a shows that, there is about $24 \sim 41$ percent performance improvement from Algorithm MK_SF_JR. For Algorithm MK_TCT_SF_JR, in contrast, it is a bi-criteria optimization algorithm for makespan and total completion time. Figs. 6a and 6b illustrate that there is a significant performance improvement ($112 \sim 132$) percent for total completion time, at the expense of a small drop in makespan improvement in comparison to Algorithm MK_SF_JR. Likewise, we will show for Facebook workloads that there can also be a very seriously negative impact in total completion time for Algorithm MK_SF_JR, whereas Algorithm MK_TCT_SF_JR can overcome and improve it significantly.



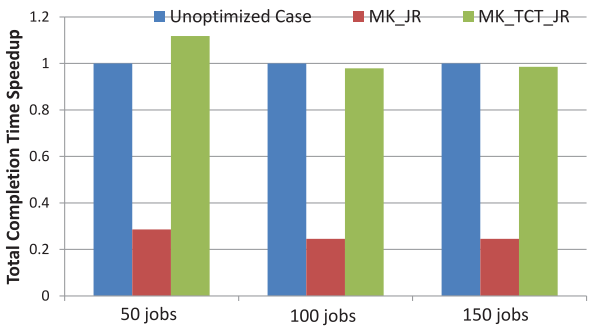
(a) *Makespan speedup* under different number of jobs



(b) *Total completion time speedup* under different number of jobs

Fig. 6. Normalized experimental results for testbed workloads under different map/reduce slot configurations and job submission orders.

(a) *Makespan speedup* under different number of jobs



(b) *Total completion time speedup* under different number of jobs

Fig. 7. Normalized simulation results for synthetic Facebook workloads under different job submission orders in a Hadoop cluster with $|\mathcal{S}^{\mathcal{M}}| = 57$ and $|\mathcal{S}^{\mathcal{R}}| = 19$.

7.2 Simulation Result with Synthetic Facebook Workload

Table 3 gives a description of MapReduce jobs in production at Facebook in October 2009, provided by Zaharia et al. [43]. They are classified into nine bins based on job sizes (numbers of maps). We make our synthetic workloads (**SW** for short) by picking representative sizes and number of jobs from each bin based on the percentage of the total number of jobs as well as the size of **SW**. For example, Table 3 shows a detailed job distribution for **SW** of 50 jobs (last two columns). For each job, we set the number of reduce tasks to 5 and 25 percent of the number of map tasks, in consistent with [43]. We estimate the running time of map and reduce tasks per job based on the map and reduce durations in Fig. 1 of [44]. More precisely, we follow the LogNormal distribution [4] with *LN*(9.9511,1.6764) for map task duration and *LN* (12.375,1.6262) for reduce task duration that fits best the Facebook task duration, given and demonstrated by [40]. Moreover, two other synthetic workloads of 100 and 150 jobs are generated by doubling and tripling the number of jobs in each bin (#Jobs in SW) in Table 3 accordingly.

TABLE 3
The Job Size Distribution at Facebook (from Table 2 in [43]) and Sizes and Number of Jobs Chosen for Each Bin in Our Synthetic Workload (**SW** for Short) of 50 Jobs

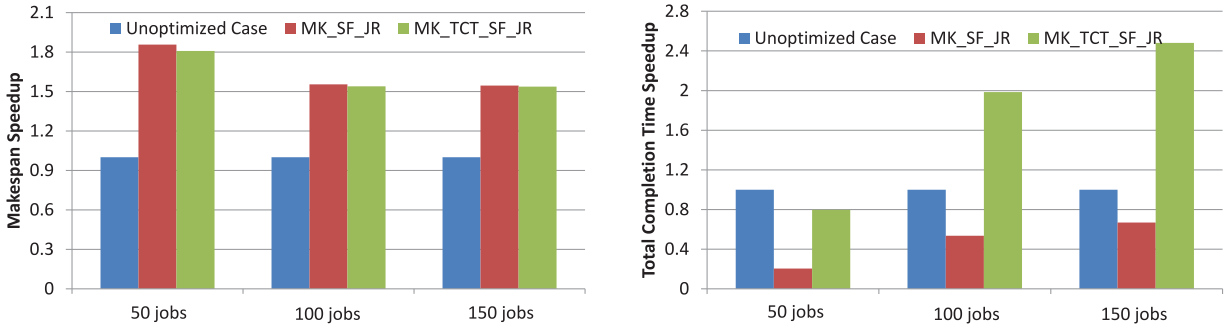
Bin	#Maps	% at Facebook	Size in SW	#Jobs in SW
0	1-25	58%	1-25	29
1	25-50	9.6%	25, 30, 35, 40, 50	5
2	50-100	8.6%	60, 80, 90, 100	4
3	100-200	8.4%	120, 150, 180, 200	4
4	200-400	5.6%	250, 320, 400	3
5	400-800	4.3%	600, 800	2
6	800-1600	2.5%	1,200	1
7	1,600-3,200	1.3%	2,400	1
8	$>$3,200	1.7%	4,800	1

7.2.1 Job Ordering Optimization Algorithms

To evaluate job ordering algorithms with respect to the synthetic Facebook workload, we use *MREstimator* to compute the makespan as well as total completion time. Fig. 7 presents the simulation results for synthetic Facebook workloads under the job ordering optimization algorithms *MK_JR* and *MK_TCT_JR*. There are about $15 \sim 19$ percent makespan improvement for *MK_JR* and $10 \sim 15$ percent for *MK_TCT_JR*. It is worth noting that the total completion time speedup in Fig. 7b for *MK_JR* is very bad, since there is a large number of small-size jobs that are put in the middle or at tail of the job list based on the rule of *MK_JR*, resulting in a big contribution to the total completion time. Compared with *MK_JR*, we can improve the total completion time significantly (about $5\times$) at the expense of a bit drop of makespan improvement for *MK_TCT_JR*.

7.2.2 Slot Configuration Optimization Algorithms

Fig. 8 shows the simulation results for synthetic Facebook workloads under varied map/reduce slot configuration optimization algorithms. There are about $55 \sim 85$ percent for Algorithm MK_SF_JR, and $54 \sim 80$ percent for Algorithm MK_TCT_SF_JR, respectively. However, Fig. 8b



(a) *Makespan speedup* under different number of jobs

(b) *Total completion time speedup* under different number of jobs

Fig. 8. Normalized simulation results for synthetic Facebook workloads under different map/reduce slot configurations and job submission orders.

illustrates that the total completion time speedup for Algorithm MK_SF_JR can be very poor when optimizing the makespan only. In comparison, there is a significant performance improvement (about $4\times$) for the total completion time at the expense of a big drop of makespan improvement for Algorithm MK_TCT_SF_JR, in comparison to Algorithm MK_SF_JR, demonstrating the importance of Algorithm MK_TCT_SF_JR that optimizes two metrics simultaneously for those workloads which contains lots of small jobs like Facebook workloads.

8 Conclusion

This paper focuses on the job ordering and map/reduce slot configuration issues for MapReduce production workloads that run periodically in a data warehouse, where the average execution time of map/reduce tasks for a MapReduce job can be profiled from the history run, under the FIFO scheduling in a Hadoop cluster. Two performance metrics are considered, i.e., makespan and total completion time. We first focus on the makespan. We propose job ordering optimization algorithm and map/reduce slot configuration optimization algorithm. We observe that the total completion time can be poor subject to getting the optimal makespan, therefore, we further propose a new greedy job ordering algorithm and a map/reduce slot configuration algorithm to minimize the makespan and total completion time together. The theoretical analysis is also given for our proposed heuristic algorithms, including approximation ratio, upper and lower bounds on makespan. Finally, we conduct extensive experiments to validate the effectiveness of our proposed algorithms and their theoretical results.

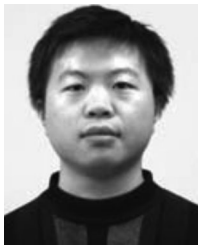
Acknowledgements

The authors thank the anonymous reviewers for their constructive comments. They acknowledge the support from the Singapore National Research Foundation under its Environmental & Water Technologies Strategic Research Programme and administered by the Environment & Water Industry Programme Office (EWI) of the PUB, under project 1002-IRIS-09.

References

[1] Amazon ec2 [Online]. Available: http://aws.amazon.com/ec2, 2015.
[2] Apache hadoop [Online]. Available: http://hadoop.apache.org, 2015.
[3] Howmanymapsandreduces [Online]. Available: http://wiki.apache.org/hadoop/HowMany MapsAndReduces, 2014.
[4] Lognormal distribution [Online]. Available: http://en.wikipedia.org/wiki/Log-normal_distribution, 2015.
[5] The scheduling problem [Online]. Available: http://riot.ieor.berkeley.edu/Applications/Scheduling/algorithms.html, 1999.
[6] S. R. Hejazi and S. Saghafian, "Flowshop-scheduling problems with makespan criterion: A review," *Int. J. Production Res.*, vol. 43, no. 14, pp. 2895–2929, 2005.
[7] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proc. 9th USENIX Conf. Netw. Syst. Design Implementation*, 2012, p. 21.
[8] P. Agrawal, D. Kifer, and C. Olston, "Scheduling shared scans of large data files," *Proc. VLDB Endow.*, vol. 1, no. 1, pp. 958–969, Aug. 2008.
[9] W. Cirne and F. Berman, "When the herd is smart: Aggregate behavior in the selection of job request," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 2, pp. 181–192, Feb. 2003.
[10] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online," in *Proc. 7th USENIX Conf. Netw. Syst. Design Implementation*, 2010, p. 21.
[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proc. 6th Conf. Symp. Oper. Syst. Design Implementation*, 2004, vol. 6, p. 10.
[12] J. Dittrich, J.-A.-Quiané Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad, "adoop++: Making a yellow elephant run like a cheetah (without it even noticing)," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 515–529, Sep. 2010.
[13] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram, "Bi-criteria algorithm for scheduling jobs on cluster platforms," in *Proc. 16th Annu. ACM Symp. Parallelism Algorithms Archit.*, 2004, pp. 125–132.
[14] P.-F. Dutot, G.Mounié, and D. Trystram,"Scheduling parallel tasks: Approximation algorithms," in *Handbo ok of Scheduling: Algorithms, Models, and Performance Analysis*, J. T. Leung, Ed. Boca Raton, FL, USA: CRC Press, ch. 26, pp. 26-1–26-24.
[15] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata, "Column-oriented storage techniques for mapreduce," *Proc. VLDB Endowment*, vol. 4, no. 7, pp. 419–429, Apr. 2011.
[16] J. Gupta, A. Hariri, and C. Potts, "Scheduling a two-stage hybrid flow shop with parallel machines at the first stage," *Ann. Oper. Res.*, vol. 69, pp. 171–191, 1997.
[17] J. N. D. Gupta, "Two-stage, hybrid flowshop scheduling problem," *J. Oper. Res. Soc.*, vol. 39, no. 4, pp. 359–364, 1988.
[18] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *Proc. VLDB Endowment*, vol. 4, no. 11, pp. 1111–1122, 2011.
[19] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *Proc. 5th Conf. Innovative Data Syst. Res.*, 2011, pp. 261–272.
[20] S. Ibrahim, H. Jin, L. Lu, B. He, and S. Wu, "Adaptive disk I/O scheduling for mapreduce in virtualized environment," in *Proc. Int. Conf. Parallel Process.*, Sep. 2011, pp. 335–344.
[21] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," *Proc. VLDB Endowment*, vol. 3, nos. 1–2, pp. 472–483, Sep. 2010.
[22] S. M. Johnson, "Optimal two- and three-stage production schedules with setup times included," *Naval Res. Logistics Quart.*, vol. 1, no. 1, pp. 61–68, 1954.

[23] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Proc. 21st Annu. ACM-SIAM Symp. Discrete Algorithms*, 2010, pp. 938–948.
[24] G. J. Kyparisis and C. Koulamas, "A note on makespan minimization in two-stage flexible flow shops with uniform machines," *Eur. J. Oper. Res.*, vol. 175, no. 2, pp. 1321–1327, 2006.
[25] J. Leung, L. Kelly, and J. H. Anderson, *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. Boca Raton, FL, USA: CRC Press, 2004.
[26] B. Moseley, A. Dasgupta, R. Kumar, and T.Sarlós, "On scheduling in map-reduce and flow-shops," in *Proc. 23rd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 289–298.
[27] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas, "Mrshare: Sharing across multiple queries in mapreduce," *Proc. VLDB Endowment*, vol. 3, nos. 1/2, pp. 494–505, Sep. 2010.
[28] C. Oğuz, and M. F. Ercan, "Scheduling multiprocessor tasks in a two-stage flow-shop environment," *Comput. Ind. Eng.*, vol. 33, nos. 3/4, pp. 269–272, Dec. 1997.
[29] C. Ouz, M. F. Ercan, T. E. Cheng, and Y. Fung, "Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop," *Eur. J. Oper. Res.*, vol. 149, no. 2, pp. 390–403, 2003.
[30] J. Polo, Y. Becerra, D. Carrera, M. Steinder, I. Whalley, J. Torres, and E. Ayguade, "Deadline-based mapreduce workload management," *IEEE Trans. Netw. Service Manage.*, vol. 10, no. 2, pp. 231–244, Jun. 2013.
[31] C. Rajendran, "Two-stage flowshop scheduling problem with bicriteria," *J. Oper. Res. Soc.*, vol. 43, no. 9, pp. 871–884, 1992.
[32] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves, "Sailfish: A framework for large scale data processing," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 4:1–4:14.
[33] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat, "Themis: An I/O-efficient mapreduce," in *Proc. 3rd ACM Symp. Cloud Comput.*, 2012, pp. 13:1–13:14.
[34] P. Sanders and J. Speck, "Efficient parallel scheduling of malleable tasks," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2011, pp. 1156–1166.
[35] S. Tang, B.-S. Lee, and B. He, "Dynamic slot allocation technique for mapreduce clusters," in *Proc. IEEE Int. Conf. Cluster Comput.*, Sep. 2013, pp. 1–8.
[36] S. Tang, B.-S. Lee, and B. He,, "Mrorder: Flexible job ordering optimization for online mapreduce workloads," in *Proc. 19th Int. Conf. Parallel Process.*, 2013, pp. 291–304.
[37] S. Tang, B.-S. Lee, and B. He, "Dynamicmr: A dynamic slot allocation optimization framework for mapreduce clusters," *IEEE Trans. Cloud Comput.*, vol. 2, no. 3, pp. 333–347, Jul. 2014.
[38] S. Tang, B.-S. Lee, and B. He, "Towards economic fairness for big data processing in pay-as-you-go cloud computing," in *Proc. IEEE 6th Int. Conf. Cloud Comput. Technol. Sci.*, Dec. 2014, pp. 638–643.
[39] S. Tang, B.-S. Lee, B. He, and H. Liu, "Long-term resource fairness: Towards economic fairness on pay-as-you-use computing systems," in *Proc. 28th ACM Int. Conf. Supercomput.*, 2014, pp. 251–260.
[40] A. Verma, L. Cherkasova, and R. H. Campbell, "Play it again, simmr!" in *Proc. IEEE Int. Conf. Cluster Comput.*, 2011, pp. 253–261.
[41] A. Verma, L. Cherkasova, and R. H. Campbell, "Two sides of a coin: Optimizing the schedule of mapreduce jobs to minimize their makespan and improve cluster performance," in *Proc. IEEE 20th Int. Symp. Model., Anal. Simul. Comput. Telecommun. Syst.*, 2012, pp. 11–18.
[42] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. balmin, "Flex: A slot allocation scheduling optimizer for mapreduce workloads," in *Proc. ACM/IFIP/USENIX 11th Int. Conf. Middleware*, 2010, pp. 1–20.
[43] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," EECS Dept., Univ. California, Berkeley, CA, USA, Tech. Rep. UCB/EECS-2009-55, Apr. 2009.
[44] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. 5th Eur. Conf. Comput. Syst.*, 2010, pp. 265–278.



Shanjiang Tang received the PhD degree from School of Computer Engineering, Nanyang Technological University, Singapore in 2015, and the MS and BS degrees from Tianjin University (TJU), China, in Jan 2011 and July 2008, respectively. He is currently an assistant professor in School of Computer Science and Technology, Tianjin University, China. In 2006, he won the 'Golden Prize' in the 31th ACM/ICPC Asia Tournament of National College Students. He was awarded the 'Talents Science Award' from Tianjin University in 2007. He was with the IBM China Research Lab (CRL) in the area of performance analysis of multi-core oriented Java multi-threaded program as an intern for four months in 2009. His research interests include parallel computing, cloud computing, big data analysis, and computational biology.

Bu-Sung Lee received the BSc (Honors) and PhD degrees from the Electrical and Electronics Department, Loughborough University of Technology, United Kingdom, in 1982 and 1987, respectively. He is currently an associate professor with the School of Computer Engineering, Nanyang Technological University, Singapore. He was elected the inaugural president of Singapore Research and Education Networks (SingAREN), 2003-2007, and has been an active member of several national standards organizations, such as Board member of Asia Pacific Advanced Networks (APAN) Ltd. His research interests include computer networks protocols, distributed computing, network management, and Grid/Cloud computing.

Bingsheng He received the bachelor's degree in computer science from Shanghai Jiao Tong University (1999-2003), and the PhD degree in computer science from the Hong Kong University of Science and Technology (2003-2008). He is an assistant professor in the Division of Networks and Distributed Systems, School of Computer Engineering, Nanyang Technological University, Singapore. His research interests are high performance computing, distributed and parallel systems, and database systems.