Optimizing MapReduce Framework through Joint Scheduling of Overlapping Phases

Huanyang Zheng, Ziqi Wan, and Jie Wu
Department of Computer and Information Sciences, Temple University, USA
Email: {huanyang.zheng, ziqi.wan, jiewu}@temple.edu

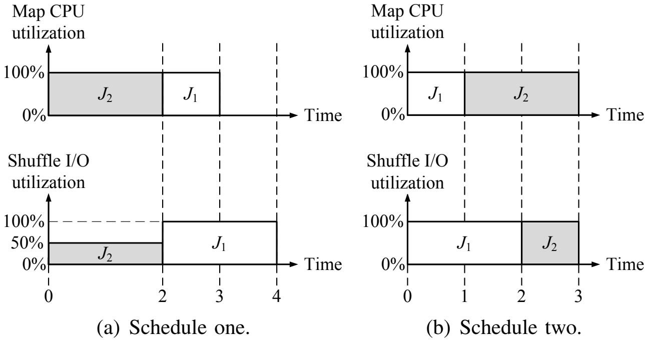
***Abstract*—MapReduce includes three phases of map, shuffle, and reduce. Since the map phase is CPU-intensive and the shuffle phase is I/O-intensive, these phases can be conducted in parallel. This paper studies a joint scheduling optimization of overlapping map and shuffle phases to minimize the average job makespan. Challenges come from the dependency relationship between map and shuffle phases, since the shuffle phase may wait to transfer the data emitted by the map phase. A new concept of the strong pair is introduced. Two jobs are defined as a strong pair, if the shuffle and map workloads of one job equal the map and shuffle workloads of the other job, respectively. We prove that, if the entire set of jobs can be decomposed to strong pairs of jobs, then the optimal schedule is to pairwisely execute jobs that can form a strong pair. Following the above intuition, several offline and online scheduling policies are proposed. They first group jobs according to job workloads, and then, execute jobs within each group through a pairwise manner. Real data-driven experiments validate the efficiency and effectiveness of the proposed policies.**

***Index Terms*—MapReduce framework, map and shuffle phases, joint scheduling, makespan optimization.**

I. INTRODUCTION

MapReduce [1] is a well-known programming framework used to process the ever-growing amount of data collected by modern instruments, such as Large Hadron Collider and next-generation gene sequencers. Although MapReduce has been widely adopted in a number of data centers, more improvements are still needed to meet the huge demands of big data computing. In the current MapReduce framework, each job consists of three dependent phases: *map*, *shuffle*, and *reduce*. The map and reduce phases typically deal with a large amount of data computations, while the shuffle phase handles the data transfer among different MapReduce workers. In terms of the resource demand, the map and reduce phases are CPU-intensive, while the shuffle phase is I/O-intensive.

Currently, most state-of-the-art research on MapReduce optimizations focuses on the map and reduce phases. However, the shuffle phase also plays an important role in transferring the data from map workers to reduce workers. It has a significant impact on the average job makespan, especially when the data is big. Moreover, Chen et al. [2] reported that jobs processed by the Facebook MapReduce cluster are shuffle-heavy. Consequently, this paper studies a joint scheduling optimization of map and shuffle phases to *minimize the average job makespan* (the time span from job arrival to shuffle phase completion). The reduce phase is not jointly optimized, since its workload is relatively light. According to [3], only 7% of jobs in a production MapReduce cluster are reduce-heavy.



(a) Schedule one. (b) Schedule two.

Fig. 1. An example for the joint scheduling of overlapping phases.

Our key observation is that the map and shuffle phases have different resource demands. Since the map phase is CPU-intensive and the shuffle phase is I/O-intensive, they can potentially be conducted in parallel to minimize the average job makespan. The key challenge comes from the fact that the map and shuffle phases cannot be fully parallelized due to their *dependency relationship*. The shuffle phase of a job must start later than its map phase, and cannot finish earlier than its map phase. This is because the shuffle phase may wait to transfer the data emitted by the map phase. An example includes the classic application of the WordCount [4], in which the map workers emit key-value pairs at a certain rate to be shuffled to the reduce workers. If the map workload of a job is larger than its shuffle workload, then the I/O resource may be underutilized, leading to a non-optimal job schedule.

To illustrate the above motivation more clearly, an example is shown in Fig. 1, which involves two jobs of J_1 and J_2. J_1 is shuffle-heavy and J_2 is map-heavy. Assuming that the resources are fully utilized, the map and shuffle phases of J_1 take 1 and 2 time slots, respectively. The resource demand of J_2 is the opposite of that of J_1 (1 time slot for the shuffle phase and 2 time slots for the map phase). As shown in Fig. 1(a), schedule one executes J_2 first, leading to an underutilization of the I/O resource. This is because J_2's shuffle phase needs to wait to transfer the data emitted by its map phase (suppose a constant data emission rate). Consequently, schedule one takes 4 time slots to finish all the jobs. As shown in Fig. 1(b), schedule two is a better scheme. It executes J_1 first, and only takes 3 time slots to finish all the jobs. It can be seen that, in order to maximally utilize the I/O resource, the shuffle-heavy job should be executed earlier than the map-heavy job.

A new concept of the strong job pair is introduced to address the above problem. Two jobs are called a *strong pair*, if the shuffle and map workloads of one job equal the map and shuffle workloads of the other job, respectively. We prove that, if the entire set of jobs can be decomposed to strong pairs of jobs, then the optimal schedule is to pairwisely execute jobs that can form a strong pair. Several offline and online scheduling algorithms are proposed to minimize the average job makespan. They first group jobs according to job workloads, and then, execute jobs within each group through a pairwise manner. Our contributions are summarized as follows:

- We address a novel MapReduce scheduling problem with respect to overlapping phases. We show that map-heavy jobs and shuffle-heavy jobs should be executed pairwisely to minimize the average job makespan.
- Four offline scheduling algorithms are proposed. Their optimalities are discussed in detail with respect to the map and shuffle workload distributions of the jobs. An online scheduling algorithm is extended.
- Real-data driven experiments are conducted to evaluate the proposed algorithms. The results are provided from different perspectives to provide insightful conclusions.

The remainder of this paper is organized as follows. Section II surveys related works. Section III formulates the problem. Section IV proposes four offline scheduling policies. Section V extends an online scheduling policy. Section VI includes experiments. Finally, Section VII concludes this paper.

II. RELATED WORK

Extensive studies on the MapReduce scheduler have been conducted over the past few years. An example includes the delay scheduling [5], which postpones the task scheduling and ameliorates the locality degradation in the Hadoop scheduler. Another example is the ARIA [6], which allocates appropriate amounts of resources to each MapReduce job to meet Service Level Objectives (SLO). Zhang et al. [7] improved ARIA by estimating the amount of resources required for completing a program. Wolf et al. [8] proposed a framework to optimize different scheduling metrics, based on a performance model, with respect to the job execution time. Tang et al. [9] proposed a scheduling policy that dynamically determines the start time of each reduce task according to its job context. Mantri [10] can mitigate the impact of outliers. It monitors task executions with real-time outlier estimations, and then takes reactions such as restarting and terminating specified outliers. Tarazu [11] was proposed as a communication-aware scheme, which schedules predictive load-balancing MapReduce jobs to reduce the network traffic within heterogeneous Hadoop clusters. Quincy [12] achieved a balanced tradeoff between the job fairness and the data locality through a min-cost flow method and a preemption mechanism. Amoeba [13] supported lightweight elastic tasks that can release the CPU resources without losing I/O computations. Multi-resource packing was investigated for schedulers [14–17]. However, the above works focus on the resource scheduling policies for map and reduce phases. The overlapping shuffle phase is not jointly optimized.

In 2013, Lin et al. [18] proposed a landmark model for the overlapping map and shuffle phases in the MapReduce. They proved that the problem of minimizing the average job makespan is NP-hard in the offline scenario, and APX-hard in the online scenario. Consequently, no online scheduling policy can guarantee a constant approximation ratio with respect to the optimal scheduling policy. However, Lin's scheduling policy may not be efficient enough, since the optimal pattern is under-explored. We show that optimal results can be obtained through pairing map-heavy jobs and shuffle-heavy jobs under load-balancing offline scenarios. Li et al. [19] considered a model with overlapping shuffle and reduce phases, utilizing the data locality to minimize the time for the shuffle phase. However, Li's scheduling policy does not fucus on the algorithmic optimality, and no approximation ratio is guaranteed. This paper is also related to Wang's research [20], where the shuffle phase is reconfigurable to dynamically coordinate the map and reduce phases. By contrast, this paper optimizes the MapReduce with a fixed shuffle workload.

III. MODEL AND PROBLEM FORMULATION

This paper focuses on a MapReduce framework with overlapping map and shuffle phases. In MapReduce, map workers continuously emit processed data (at a constant rate), which are in turn shuffled to reduce workers. We consider that map and shuffle phases mainly take CPU and I/O resources, respectively. Hence, they may be conducted in parallel. However, the shuffle phase is dependent on the map phase. This is because the shuffle phase may wait to transfer the data emitted by the map phase. If the data transfer rate of the shuffle phase is higher than the data emission rate of the map phase, then the shuffle phase has to wait for the data emission. As a result, the shuffle phase of a job must start later than its map phase, and cannot finish earlier than its map phase. The reduce phase is not jointly optimized, since its workload is light [3].

We study both *offline* and *online* scenarios with n jobs in total. The offline scenario means that all jobs arrive at the system *at the start time*, waiting to be scheduled (job information is pre-known). The online scenario means that the scheduler only obtains the workload information of a job upon its arrival, which may not be the start time. Let $J = \{J_1, J_2, ..., J_n\}$ denote the set of jobs, where J_i is the ith job. Let t_i^m and t_i^s denote the map and shuffle workloads of J_i, respectively. The workload of a job is its execution time under fully-utilized resources. A MapReduce job may include multiple parallel subtasks on different machines. In such an event, its workload is the sum among different subtasks. The CPU resource is always fully utilized. In contrast, the I/O resource may be underutilized due to the dependency relationship between map and shuffle phases. The actual shuffle time is considered to be reversely proportional to the I/O utilization. For example, when the I/O utilization is 25%, the shuffle time is quadrupled. We have the following definitions:

***Definition** 1:* The job of J_i is said to be balanced if and only if $t_i^m = t_i^s$. If $t_i^m > t_i^s$, J_i is map-heavy. On the other hand, if $t_i^m < t_i^s$, J_i is shuffle-heavy.

***Definition** 2:* The makespan of a job is the time span from its arrival to its shuffle phase completion.

Our objective is to *minimize the average job makespan* through jointly scheduling overlapping map and shuffle phases. We assume that the MapReduce has a *centralized scheduler*, which abstracts the job schedule as a *sequential order*. The scheduler executes the next job, only if the MapReduce cluster has sufficient machines with idle CPU resources. This is because the next job may require the CPU resources of multiple machines to start its map phase. Our problem is NP-hard and APX-hard in the offline and online scenarios, respectively [18].

Note that a job may not start immediately after its arrival, since it may be scheduled to wait for other jobs. To minimize the average job makespan, we prefer to execute jobs with lighter workloads earlier. This is because the smaller jobs can finish earlier. The key challenge comes from the dependency relationship between map and shuffle phases, which may lead to I/O underutilization (and thus a non-optimal schedule). As a result, the optimal schedule may not be simply ranking jobs by their workloads. The following two sections will explore more insights in offline and online scenarios, respectively.

IV. OFFLINE SCHEDULING SCENARIO

A. Pair-based Scheduling Policy and Discretization

We first present a pair-based scheduling policy. For clear presentations, the following definitions are introduced:

***Definition** 3:* Jobs J_i and J_j are a weak pair, if $t_i^m + t_j^m = t_i^s + t_j^s$. They are a strong pair, if $t_i^m = t_j^s$ and $t_i^s = t_j^m$.

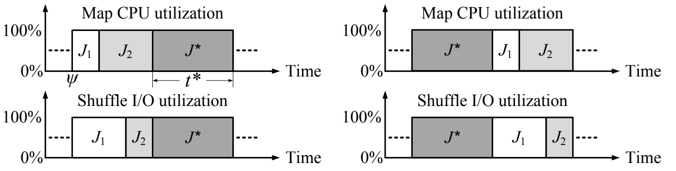
If two jobs can form a weak pair, then they can be executed together to avoid I/O underutilization. If two jobs can form a strong pair, then their map and shuffle workloads are exactly opposite to each other. A strong pair is necessarily, but not sufficiently, a weak pair. Our key result is shown as follows:

***Theorem** 1:* If J can be decomposed to strong pairs of jobs, then jobs that can form a strong pair are pairwisely executed in J's optimal schedule. For each strong pair, the shuffle-heavy job is executed before the map-heavy job.

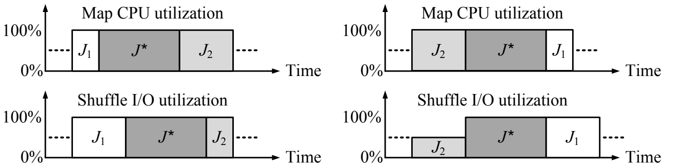
Proof: We prove by induction. Let us start with a base case, where J only includes two jobs that can form a strong pair (denoted as J_1 and J_2). Suppose J_1 is shuffle-heavy and J_2 is map-heavy. We have two schedules: schedule one executes J_1 before J_2; and schedule two executes J_2 before J_1. Then, the job makespans of J_1 and J_2 are shown as follows:

Job makespans	J_1	J_2
Schedule one (J_1 before J_2)	t_1^s	$t_1^m + t_2^m$
Schedule two (J_2 before J_1)	$t_2^m + t_1^s$	t_2^m

We have $t_1^s = t_2^m$ according to the definition of the strong pair. Since J_1 is shuffle-heavy ($t_1^m < t_1^s$), we have $t_1^m + t_2^m < t_2^m + t_1^s$. Hence, schedule one has a smaller average job makespan by executing the shuffle-heavy job before the map-heavy job.



(a) Schedule S_1. (b) Schedule S_2.



(c) Schedule S_3. (d) Schedule S_4.

Fig. 2. An illustration for the proof of Theorem 1.

For the induction, let us consider an existing schedule of S. It pairwisely executes jobs that can form a strong pair. Let J^* denote a subset of J that are consecutively and pairwisely executed in S. Let τ denote the average job makespan of J^* (but calculated from the execution time of J^*). Let t^* denote the total map workloads of J^*. Since jobs in J^* are strongly paired, t^* is also the total shuffle workloads of J^*. The induction step adds one more strong pair of jobs to schedule S (say shuffle-heavy J_1 and map-heavy J_2). As shown in Fig. 2, there exist four possible schedules to incorporate J_1 and J_2 into S: S_1 executes J_1 and J_2 before J^*; S_2 executes J_1 and J_2 after J^*; S_3 executes J_1 before J^*, and J_2 after J^*; S_4 executes J_2 before J^*, and J_1 after J^*. S_1 and S_2 execute J_1 and J_2 in a pairwise manner, while S_3 and S_4 execute J_1 and J_2 in an interwoven manner. Suppose that J_1, J_2, and J^* are scheduled at time ψ, then their job makespans are:

Job makespans	J_1	J_2	J^*
Schedule S_1	$\psi+t_1^s$	$\psi+t_1^m+t_2^m$	$\psi+\tau+t_1^m+t_2^m$
Schedule S_2	$\psi+t_1^s+t^*$	$\psi+t_1^m+t_2^m+t^*$	$\psi+\tau$
Schedule S_3	$\psi+t_1^s$	$\psi+t_1^m+t_2^m+t^*$	$\psi+\tau+t_1^s$
Schedule S_4	$\psi+t_1^s+t_2^m+t^*$	$\psi+t_2^m$	$\psi+\tau+t_2^m$

It is trivial that S_4 is always worse than S_3, due to its I/O underutilization of J_2. Meanwhile, the average job makespans of S_1, S_2, and S_3 are shown as follows:

$$\begin{aligned} S_1 : \psi + \frac{\left[|J^*|\cdot(t_1^m+t_2^m)\right] + \left[t_1^s+(t_1^m+t_2^m)+|J^*|\cdot\tau\right]}{|J^*|+2} \\ S_2 : \psi + \frac{\left[\quad 2t^* \quad\right] + \left[t_1^s+(t_1^m+t_2^m)+|J^*|\cdot\tau\right]}{|J^*|+2} \\ S_3 : \psi + \frac{\left[\ |J^*|\cdot t_1^s+t^*\ \right] + \left[t_1^s+(t_1^m+t_2^m)+|J^*|\cdot\tau\right]}{|J^*|+2} \end{aligned} \tag{1}$$

Here, $|J^*|$ denotes the number of jobs in J^*. A notable point is $|J^*| \cdot (t_1^m + t_2^m) < |J^*| \cdot 2t_1^s$ according to the definitions of J_1 and J_2. We have the following inequality:

$$\frac{|J^*|\cdot(t_1^m+t_2^m)+2t^*}{2} < \frac{|J^*|\cdot 2t_1^s+2t^*}{2} = |J^*| \cdot t_1^s + t^* \tag{2}$$

The mean of two unequal numbers is always larger than the minimal one of these two numbers. Therefore, we have:

$$\min\{|J^*|\cdot(t_1^m+t_2^m), 2t^*\} < |J^*| \cdot t_1^s + t^* \tag{3}$$

Eqs. 1 and 3 indicate that either S_1 or S_2 has the smallest average job makespan. Hence, J_1 and J_2 should be pairwisely

Algorithm 1 Pair-based Scheduling Policy

```
Input:  The job set, $J$, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.
1: Initialize an array, $S$, to represent the job execution order;
2: Put all jobs into the order array of $S$;
3: Sort all jobs in $S$ according to $\max(t_i^m, t_i^s)$;
4: for each subset of jobs with the same $\max(t_i^m, t_i^s)$ do
5:     Reorder jobs by iteratively taking out a pair of jobs of
       $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
6: return the order array of $S$ as the schedule;
```

executed, when being incorporated into S. By induction, jobs that can form a strong pair should be pairwisely executed in the optimal schedule. We also conclude that, for each strong pair, the shuffle-heavy job is executed before the map-heavy job. Therefore, the proof of Theorem 1 completes. ■

Theorem 1 shows that we could avoid I/O underutilization by pairwisely executing jobs that can form a strong pair. This idea can be extended by organizing a bundle of jobs (such as a 3-tuple of jobs) as a basic scheduling unit. However, such an extension may bring a higher scheduling complexity, and may post a higher optimality prerequisite on the workload distributions of jobs. Therefore, we use a pair of jobs (rather than a 3-tuple of jobs) as the basic scheduling unit.

We propose Algorithm 1, which has two stages. The first stage (lines 1 to 3) is based on Lin's MaxSRPT algorithm [18], where jobs are sorted according to $\max(t_i^m, t_i^s)$. Note that $\max(t_i^m, t_i^s)$ represents the dominant workload of J_i. Jobs with lighter workloads should be executed earlier, since small jobs could finish earlier to minimize the average job makespan. The second stage (lines 4 and 5) is our novel contribution based on Theorem 1. Jobs are iteratively paired according to their map and shuffle workload differences. We prioritize jobs with smaller workloads (the first stage) over jobs with better pairs (the second stage), since the former one generally rules the latter one (as verified in experiments). The time complexity of Algorithm 1 is $O(n \log n)$, in which n is the number of jobs. This time complexity results from the sorting procedure in Algorithm 1 (lines 3 and 5).

A potential problem of Algorithm 1 is on the job workload granularity. The second stage of Algorithm 1 pairs jobs with the same dominant workloads, i.e., the same $\max(t_i^m, t_i^s)$. If each job has a unique dominant workload, then the pairing process is skipped and thus becomes useless. To control the granularity, we additionally introduce a discretization process before applying Algorithm 1. Let Δ denote the discretization step, where the map and shuffle workloads of each job are rounded to the nearest multiple of Δ. A larger Δ represents a coarser workload granularity, where more jobs share the same dominant workloads. A smaller Δ brings fine-grained workloads, where less jobs share the same dominant workloads.

To better explain Algorithm 1 and the discretization process, an example is shown in Table I. It includes 6 jobs ($n = 6$) with a discretization step of $\Delta = 2$. The discretization process rounds the map and shuffle workloads of each job to multiples of Δ. Then, Algorithm 1 is applied. In the first stage, jobs are sorted according to $\max(t_i^m, t_i^s)$. J_1 and J_5 have dominant workloads of 2Δ, while J_2, J_3, J_4, and J_6 have dominant workloads of 4Δ. In the second stage, jobs are paired. J_1 and J_5 have the same dominant workloads, and are paired directly (although they are both shuffle-heavy). For the remaining four jobs, J_4 and J_2 are first paired. This is because J_4 and J_2 are shuffle-heaviest and map-heaviest, respectively. J_3 and J_6 are paired at the end. Consequently, the job execution order in the final schedule is J_5, J_1, J_4, J_2, J_3, and J_6.

TABLE I

EXAMPLE OF ALGORITHM 1.

Jobs	J_1	J_2	J_3	J_4	J_5	J_6
t_i^m	2.1	8.8	6.6	4.5	1.8	8.0
t_i^s	3.6	4.3	8.4	7.9	4.1	7.6
Discrete t_i^m	Δ	4Δ	3Δ	2Δ	Δ	4Δ
Discrete t_i^s	2Δ	2Δ	4Δ	4Δ	2Δ	4Δ
$\max(t_i^m, t_i^s)$	2Δ	4Δ	4Δ	4Δ	2Δ	4Δ

Algorithm 1 works well *when only a small portion of jobs can be paired.* Its optimality is stated as follows:

Theorem 2: Algorithm 1 is optimal, when all jobs in J are simultaneously map-heavy, balanced, or shuffle-heavy.

Proof: When all jobs in J are simultaneously map-heavy, the shuffle workload has no impact on the job makespan. This is because the I/O resource is always underutilized for each job. At this time, Algorithm 1 schedules jobs according to their map workloads. It is trivial that jobs with lighter map workloads should be executed earlier to minimize the average job makespan, since the smaller jobs can finish earlier. When all jobs in J are simultaneously balanced or shuffle-heavy, the scenario is similar, and thus, the proof completes. ■

When all jobs in J are simultaneously map-heavy, balanced, or shuffle-heavy, the dependency relationship between map and shuffle phases has no impact on the job makespan with respect to different schedules. In such a case, Algorithm 1 schedules jobs based on their dominant workloads, resulting in the scheduling optimality. The following subsection will explore another pairwise scheduling.

B. Couple-based Scheduling Policy and Generalization

The previous subsection introduced Algorithm 1 to schedule jobs in a pairwise manner. Its intuition is based on Theorem 1, where jobs should be pairwisely executed to meet the scheduling optimality under certain scenarios. However, Algorithm 1 fails to work well when a large portion of jobs can be paired. As a simple variation of Algorithm 1, Algorithm 2 is proposed to address the above issue. Similar to Algorithm 1, Algorithm 2 also has two stages. In the first stage (lines 1 to 3), all jobs are sorted according to their total map and shuffle workloads, i.e., $t_i^m + t_i^s$. Its intuition is similar to that of Algorithm 1: jobs with lighter workloads should be executed earlier, since the smaller jobs can finish earlier to minimize the average job makespan. The key difference is that jobs are sorted by total map and shuffle workloads in Algorithm 2,

Algorithm 2 Couple-based Scheduling Policy

```
Input: The job set, $J$, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.
1: Initialize an array, $S$, to represent the job execution order;
2: Put all jobs into the order array of $S$;
3: Sort all jobs in $S$ according to $t_i^m + t_i^s$;
4: for each subset of jobs with the same $t_i^m + t_i^s$ do
5:     Reorder jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
6: return the order array of $S$ as the schedule;
```

instead of dominant workloads in Algorithm 1. The second stage of Algorithm 2 (lines 4 and 5) is identical to Algorithm 1, where jobs are iteratively paired based on their map and shuffle workload differences. The time complexity of Algorithm 2 remains $O(n \log n)$ for the same reason as Algorithm 1.

Algorithm 2 works well *when a large portion of jobs can be paired*. Its optimality is stated as follows:

***Theorem** 3:* Algorithm 2 is optimal, when J can be decomposed to strong pairs of jobs.

Proof: The proof starts with constructing a new job set of J' from J. Each strong pair of jobs in J (say J_i and J_j) is mapped to a job in J'. The mapped job in J' has map and shuffle workloads of $t_i^m + t_j^m$ and $t_i^s + t_j^s$, respectively. By the definition of the strong pair, we have $t_i^m + t_j^m = t_i^s + t_j^s$. Therefore, each job in J' is balanced. Basically, J' is constructed by merging each strong pair of jobs in J. According to Theorem 1, when J can be decomposed to strong pairs of jobs, jobs that can form a strong pair are pairwisely executed in the optimal schedule of J. Consequently, the optimal schedule for J' is the same as the optimal schedule for J. While each job in J' is balanced, it is trivial that jobs with lighter workload should be executed earlier to minimize the average job makespan, since the smaller jobs can finish earlier. If a job with a heavier workload is executed before a job with a lighter workload, then a swap of their execution order always leads to a smaller average job makespan. Hence, Algorithm 2 is optimal, when J can be decomposed to strong pairs of jobs. ■

The key insight behind Theorem 3 is to achieve an optimal pairwise schedule by considering the total workload of a pair of jobs, instead of the workload of a single job. Note that Algorithms 1 and 2 are equivalent to each other, when all jobs in J are simultaneously balanced. This is because $t_i^m + t_i^s$ is proportional to $\max(t_i^m, t_i^s)$ when $t_i^m = t_i^s$. Hence, we have:

***Corollary** 1:* Algorithms 1 and 2 are equivalent and optimal, when all jobs in J are simultaneously balanced.

Algorithm 3 Generalized Scheduling Policy

```
Input: The job set, $J$, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.
Initialize an array, $S$, to represent the job execution order;
Put all jobs into the order array of $S$;
Set $J_i$'s priority as $[\alpha\cdot\max(t_i^m, t_i^s) + (1-\alpha)\cdot(t_i^m + t_i^s)]$;
Sort all jobs in $S$ according to their priorities;
for each subset of jobs with the same priority do
    Reorder jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
return the order array of $S$ as the schedule;
```

While Algorithm 1 works well when few jobs can be paired, Algorithm 2 works well when many jobs can be paired. They are equivalent and optimal when all jobs are balanced. To resolve the above tradeoff, Algorithm 3 is proposed to combine Algorithms 1 and 2. It uses $[\alpha\cdot\max(t_i^m, t_i^s) + (1-\alpha)\cdot(t_i^m + t_i^s)]$ as J_i's priority, and then sort all jobs according to their priorities. α is a weight parameter that satisfies $0 \leq \alpha \leq 1$. Algorithm 3 reduces to Algorithm 1 when $\alpha = 1$, and reduces to Algorithm 2 when $\alpha = 0$. The job priority in Algorithm 3 is a weighted combination of those in Algorithms 1 and 2. These three algorithms have the same time complexity of $O(n \log n)$, which comes from the sorting procedure. In addition, they all rely on the discretization process to control the job granularity, such that jobs with similar priorities are grouped for the pairing process. However, the discretization process is not necessary and can be replaced by some other methods. The following subsection will present the details.

C. Group-based Scheduling Policy

Previous subsections introduced Algorithms 1, 2, and 3 to schedule jobs with a discretization process, which controls the granularity of the job priority. Jobs with similar priorities are grouped for the pairing process. The discretization process is essentially a grouping (or clustering) procedure, and thus, it could be replaced by other grouping methods. This subsection presents a pairwise scheduling policy that groups jobs through a dynamic programming approach. The grouping goal is to divide jobs to k (a pre-specified parameter) groups, such that the Sum of Maximum Job Priority Difference within each group (SMJPD) is minimized. Let $G_1, G_2, ..., G_k$ denote the k job groups. Then, SMJPD can be computed as follows:

$$\text{SMJPD} = \sum_{l=1}^{k} \Big\{ \max_{J_i, J_j \in G_l} \Delta_{i,j} \Big\} \tag{4}$$

$$\begin{aligned}\Delta_{i,j} = \big| &[\alpha\cdot\max(t_i^m, t_i^s) + (1-\alpha)\cdot(t_i^m + t_i^s)] \\ &-[\alpha\cdot\max(t_j^m, t_j^s) + (1-\alpha)\cdot(t_j^m + t_j^s)]\big|\end{aligned} \tag{5}$$

Here, $\Delta_{i,j}$ denotes the job priority difference between J_i and J_j. The optimal grouping result can be obtained by a dynamic programming approach. Without loss of generality, we assume that all jobs are already sorted according to their priorities, i.e., $[\alpha\cdot\max(t_i^m, t_i^s) + (1-\alpha)\cdot(t_i^m + t_i^s)]$ is non-decreasing with respect to the index i. Let $OPT_{j,l}$ denote the optimal SMJPD for the first j jobs $(J_1, J_2, ..., J_j)$, when they are divided to l groups. $OPT_{n,k}$ is the desired result. The optimal substructure for the dynamic programming approach is shown as follows:

$$OPT_{j,l} = \min_{l \leq i \leq j} \{OPT_{i-1,l-1} + \Delta_{i,j}\} \tag{6}$$

Since jobs are assumed to be sorted by their priorities, $\Delta_{i,j}$ is also the maximum job priority difference for the job group of

Algorithm 4 Group-based Scheduling Policy

```
Input:  The job set, $J$, and its workloads, $\{t_i^m\}$ and $\{t_i^s\}$.
Output: A schedule of the job execution order.
1: Initialize an array, $S$, to represent the job execution order;
2: Put all jobs into the order array of $S$;
3: Set $J_i$'s priority as $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$;
4: Sort all jobs in $S$ according to their priorities;
5: Divide jobs into $k$ groups by dynamic programming:
       Initialize a two-dimensional array of $OPT$;
       Initialize $OPT_{j,l} = 0$ when $j = 0$ or $l = 0$;
       Compute $OPT_{j,l} = \min_{l \leq i \leq j}\{OPT_{i-1,l-1} + \Delta_{i,j}\}$;
       Trace back the optimal job grouping through index $i$;
6: for each group of jobs do
7:     Reorder jobs by iteratively taking out a pair of jobs of
       $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$;
8: return the order array of $S$ as the schedule;
```

$J_i, J_{i+1}, ..., J_j$. Then, Eq. 6 can be interpreted as follows. The optimal grouping for the first j jobs of l groups is composed of (1) the optimal grouping for the first $i-1$ jobs of $l-1$ groups, and (2) the remaining jobs of $J_i, J_{i+1}, ..., J_j$ as a new group. The index of i is traversed to guarantee the optimality. Since i is traversed, computing the dynamic programming entry of $OPT_{j,l}$ takes $O(n)$ on average. There exist $O(nk)$ entries in total, and thus, the eventual time complexity of the dynamic programming approach is $O(n^2k)$. As for the initialization, we can simply set $OPT_{j,l} = 0$ when $j = 0$ or $j \leq l$.

Algorithm 4 is proposed as a scheduling policy that groups jobs through the above technique. Similar to Algorithm 3, it uses $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$ as the job priority of J_i (lines 1 to 3). Then, the dynamic programming approach is applied to group jobs based on their priorities. Meanwhile, groups are also sorted according to their priority range (lines 4 and 5). For each group, jobs are iteratively paired according to their map and shuffle workload differences (lines 6 and 7). We prioritize jobs with smaller workloads (the first stage) over jobs with better pairs (the second stage), since the former one generally rules the latter one (as verified in experiments). The time complexity of Algorithm 4 is $O(n^2k)$, which results from the dynamic programming approach. Although Algorithm 4 has a higher time complexity than Algorithms 1, 2, and 3, it skips the discretization process that may result in information loss. As a tradeoff, Algorithm 4 controls the job granularity through a more flexible manner in terms of the parameter k.

V. ONLINE SCHEDULING SCENARIO

This section studies the online scenario, where jobs are no longer known a priori. The scheduler can only obtain the workload information of a job upon its arrival. Due to the lack of knowledge on future job arrivals, the online scheduling problem is harder. The online scheduling problem is APX-hard according to Lin's report [18], meaning that no online algorithm can guarantee a constant approximation ratio with respect to different job arrivals. Consequently, we propose a heuristic scheduling algorithm based on Algorithm 4.

Algorithm 5 Online Group-based Scheduling Policy

```
Input:  The old schedule, $S$, and a new arriving job, $J_i$.
Output: A new schedule of the current job execution order.
1: Set $J_i$'s priority as $[\alpha \cdot \max(t_i^m, t_i^s) + (1-\alpha) \cdot (t_i^m + t_i^s)]$;
2: if $rand() < 1/nk^2$ then
3:     Call Algorithm 4 to completely reschedule all jobs;
4:     return the new schedule;
5: else
6:     for each job group, $G_l$, in $S$ do
7:         Compute $\max_{J_j \in G_l} \Delta_{i,j}$;
8:     Add $J_i$ into $G_l = \arg\min_{G_l}\{\max_{J_j \in G_l} \Delta_{i,j}\}$;
9:     Reorder jobs in $G_l$ via the same way as Algorithm 4;
10:    return the updated $S$ as the schedule;
```

The proposed online scheduling algorithm includes an initialization process. At the system start time, Algorithm 4 is used to schedule the existing jobs. If the number of existing jobs is less than k, then each job is regarded as a job group. Note that job groups are sorted according to their priority ranges. Upon a new job arrival, Algorithm 5 is called. It includes two sub-methods: method one completely reschedules all jobs (lines 2 and 3), and method two slightly modifies the existing old schedule (lines 5 to 10). Methods one and two are chosen through a random number generator of $rand()$ in line 1. The function of $rand()$ returns a uniformly random number between 0 and 1. Therefore, line 1 indicates that, Algorithm 5 has a small probability of $\frac{1}{nk^2}$ to choose method one, and has a large probability of $1 - \frac{1}{nk^2}$ to choose method two. Here, n is the total number of jobs that are waiting for the schedule. The above probabilities aim to balance the time complexity.

Method one calls Algorithm 4 to reschedule all jobs, and thus takes a time complexity of $O(n^2k)$. In contrast, method two modifies the existing old schedule to resolve the new job. It checks every job group for the new arrival job, and then adds the new job to its closest existing job group. The closest group is the one that can minimize the maximum job priority difference with the new job (lines 6 to 8). It can be found within a time complexity of $O(k)$, since we only need to check the minimum and maximum job priority in each job group. All jobs in this group and the new arrival job are completely reordered in a pairwise manner (line 9). Since each job group is expected to include $\frac{n}{k}$ jobs, method two is also expected to take $O(\frac{n}{k})$. Consequently, Algorithm 5 takes $O(\frac{n}{k})$, since $[\frac{1}{nk^2} \cdot O(n^2k) + (1 - \frac{1}{nk^2}) \cdot O(\frac{n}{k})] \in O(\frac{n}{k})$.

Although the online scheduling problem is APX-hard, Algorithm 5 could be optimal when all jobs are balanced. In such an event, Algorithm 5 is reduced to scheduling jobs according to their workloads, where jobs with lighter workloads should be executed earlier. The key idea of Algorithm 5 is to balance the scheduling performance and time complexity through two methods. Method one has a better scheduling performance at the cost of a larger time complexity, while method two has a worse scheduling performance but a smaller time complexity. They are balanced through the random number generator.

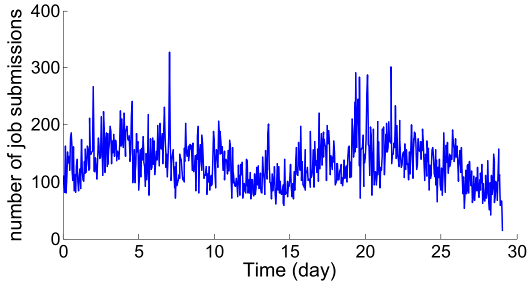
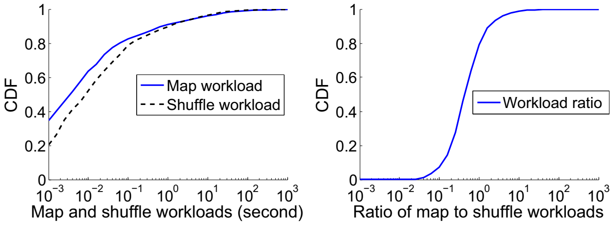


Fig. 3. Number of job submissions per hour.



(a) Map and shuffle workloads. (b) Workload ratio distribution.

Fig. 4. Job workloads in Google cluster (cumulative distribution functions).

VI. EXPERIMENTS

We conduct real data-driven experiments to evaluate the performance of the proposed offline and online scheduling algorithms. The evaluation results are shown from different perspectives to provide insightful conclusions.

A. *Google Cluster Dataset*

Our major offline and online experiments are conducted based on the Google cluster dataset [21, 22]. It is a real workload trace of a Google cluster with about 11,000 machines. It spans 29 days in May 2011. The total size of the compressed trace is approximately 41GB. This trace recorded all the detailed information about job arrivals and resource usage for each task with time stamps, in milliseconds. MapReduce jobs are filtered out through some eliminations (e.g., jobs with a single task, jobs without CPU or disk usage). As a result, we obtain 96,182 jobs over 29 days. The job submission rate per hour is shown in Fig. 3. It has two slight peaks (around days 3 to 7 and days 18 to 22). The average job submission rate is 138 jobs per hour, while the highest rate is more than 300 jobs per hour around day 7. It can be seen that there are diurnal and weekly job submission patterns in the trace. This is because people work more during the daytime than at night, as well as weekdays over weekends. Fig. 4(a) shows the map and shuffle workload distributions in the Google cluster dataset. About 70% of map and shuffle workloads are less than 1 second. Fig. 4(b) shows the distribution of the ratio of map workload to shuffle workload per job. The ratios of majority jobs range from 0.1 to 10. Very few jobs have map-to-shuffle workload ratios that are smaller than 0.1 or larger than 10. Total workloads are balanced. Fig. 4(b) means that a large portion of jobs in the Google cluster can be paired. Consequently, our algorithms are applicable.

B. *Comparison Algorithms and Metrics*

The following four algorithms are used for comparison:

- MaxDiff ranks jobs by their map and shuffle workload differences (i.e., $t_i^m - t_i^s$ for job J_i). The job with a larger workload difference is executed later. The motivation is that, it prioritizes shuffle-heavy jobs over map-heavy jobs to avoid I/O resource underutilization.
- Pairwise is based on Theorem 1, which suggests that jobs should be pairwisely scheduled. This policy orders jobs by iteratively taking out a pair of jobs of $J_i = \arg\max_i(t_i^s - t_i^m)$ and $J_j = \arg\max_j(t_j^m - t_j^s)$.
- MaxShuffle ranks jobs by their shuffle workloads. Jobs with a larger shuffle workload are executed earlier, in order to avoid I/O resource underutilization.
- MaxSRPT is proposed by Lin et al. [18]. It schedules jobs according to their dominant workload (i.e., $\max(t_i^m, t_i^s)$ for job J_i). Our algorithms improve MaxSRPT through executing jobs pairwisely, according to Theorem1.

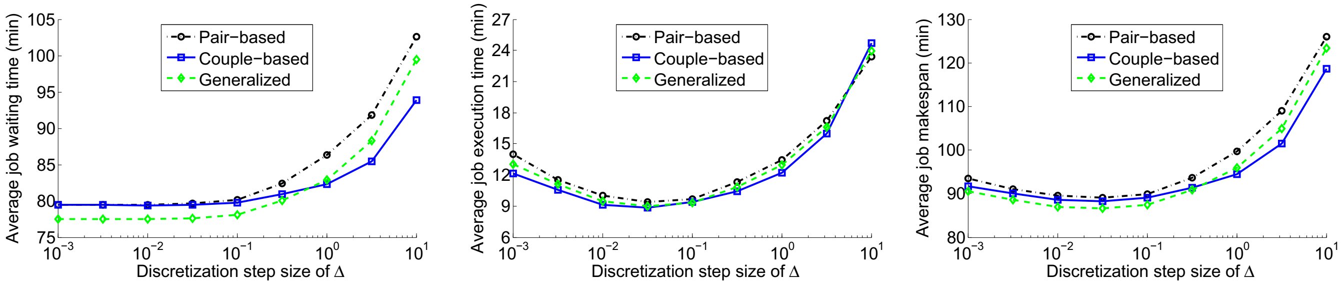
In addition, experiments present Algorithms 1 to 5 as Pair-based, Couple-based, Generalized, Group-based, and Online group-based scheduling policies for simplicity. In default, we use $\Delta = 0.1$ seconds as the discretization step (Algorithms 1 to 3), $\alpha = 0.5$ as the weight parameter (Algorithms 3 to 5), and $k = 20$ as the number of groups (Algorithms 4 and 5).

Three metrics are used for comparison. The first metric is the average job makespan, which is the time span from the job arrival to its shuffle phase completion. The other two metrics are the *average job waiting time* and the *average job execution time*. The waiting time of a job is the time span from the job arrival to its map phase start. The execution time of a job is the time span from its map phase start to its shuffle phase completion. By definition, the job makespan is the sum of the job waiting time and the job execution time.

C. *Evaluation Results for Offline Scheduling*

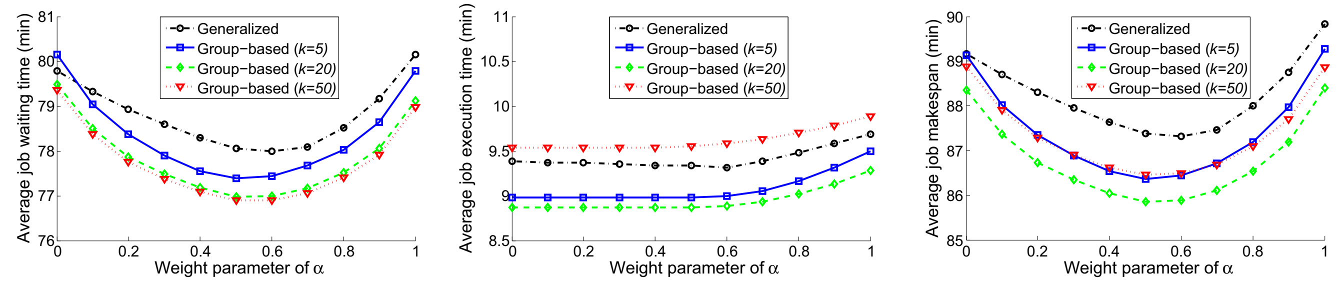
Experiments in the Google cluster dataset are conducted for the offline scenario, where all jobs are supposed to arrive at the system start time. The results are shown in Table II with the unit of seconds. MaxDiff, Pairwise, and MaxShuffle have the worst performance. However, Pairwise has a significant smallest average job execution time through executing jobs pairwisely. It ignores the total map and shuffle workloads of jobs, leading to an overly large job waiting time. We can also find that Pair-based scheduling policy has a larger average job wait time than MaxSRPT, since the discretization process is information-lossy. However, the former policy has a smaller average job execution time through executing jobs pairwisely. Couple-based policy improves Pair-based policy through considering the total map and shuffle workloads of a job rather than its dominant workload. Generalized policy improves Pair-based and Couple-based policies by combining them with a weight parameter of α. Group-based policy improves Generalized policy by grouping jobs optimally.

The impacts of the discretization step size, Δ, the weight parameter, α, and the group number, k, are shown in Figs. 5



(a) Average job waiting time. (b) Average job execution time. (c) Average job makespan.

Fig. 5. Offline performance evaluation with respect to the discretization step size of Δ.



(a) Average job waiting time. (b) Average job execution time. (c) Average job makespan.

Fig. 6. Offline performance evaluation with respect to the weight parameter α.

TABLE II

OFFLINE PERFORMANCE EVALUATION IN GOOGLE CLUSTER DATASET.

Scheduling algorithms	Average job waiting time	Average job execution time	Average job makespan
MaxDiff	8806	682	9488
Pairwise	8289	149	9138
MaxShuffle	7929	898	8827
MaxSRPT	4768	840	5608
Pair-based	4809	581	5390
Couple-based	4787	563	5350
Generalized	4683	560	5243
Group-based	4619	532	5151

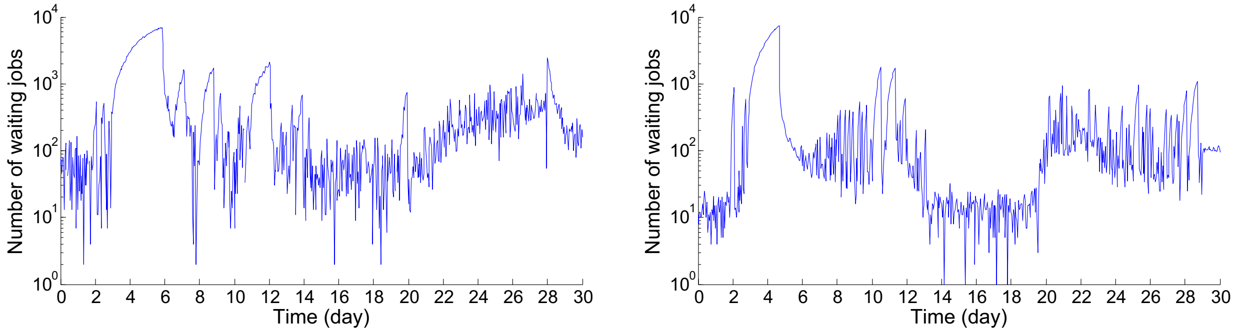
and 6 (offline scenario in the Google cluster dataset). Fig. 5(a) shows that a small Δ does not have a significant impact on the average job waiting time. However, a large Δ results in an exponentially increased average job waiting time, due to the information loss on the total or dominant job workload. Meanwhile, Fig. 5(b) shows that both overly small and overly large Δ will increase the job execution time. This is because the pairing process is broken down by an improper Δ. The corresponding average job makespan is shown in Fig. 5(c). As for the weight parameter α, Fig. 6(a) shows an interesting pattern. Generalized policy reduces to Pair-based policy when $\alpha = 1$, and reduces to Couple-based policy when $\alpha = 0$. However, it achieves the smallest average job waiting time when α is around 0.6. Meanwhile, α has a slight impact on the average job execution time. Another notable point is with respect to k. While Fig. 6(a) shows that an overly small k leads to a large average job wait time, Fig. 6(b) shows that an overly large k leads to a large average job execution time. As shown in Fig. 6(c), in order to minimize the average job makespan, k should be neither too small nor too large.

D. *Evaluation Results for Online Scheduling*

Experiments in the online scenario are conducted in the Google cluster dataset, which includes the job arrival time. Previous offline algorithms are applied in the online scenario through completely rescheduling all existing jobs upon each new job arrival. We start with the number of waiting jobs per hour under each scheduling policy. The results are shown in Fig. 7. Not all policies are presented here due to the page limitation. Fig. 7(a) focuses on the Pairwise policy, which has the worst performance. Compared to other policies, Pairwise has a larger number of waiting jobs for a longer time around days 4, 5, and 12. It also has more waiting jobs from days 22 to 30. Fig. 7(b) shows the result for MaxSRPT, which is not the best one, due to the peak for days 20 to 24. In contrast, Group-based scheduling policy has the smallest number of waiting jobs over time, as shown in Fig. 7(c). This is because it considers to schedule jobs in a pairwise manner to avoid the underutilization of the I/O resource. The performance of Online group-based scheduling policy is shown in Fig. 7(d). It has a slightly worse performance from days 26 to 30 than its offline version. Note that the scheduling time complexity of the online version is $O(\frac{n}{k})$, which is lower than the scheduling time complexity of the offline version, $O(n^2k)$.

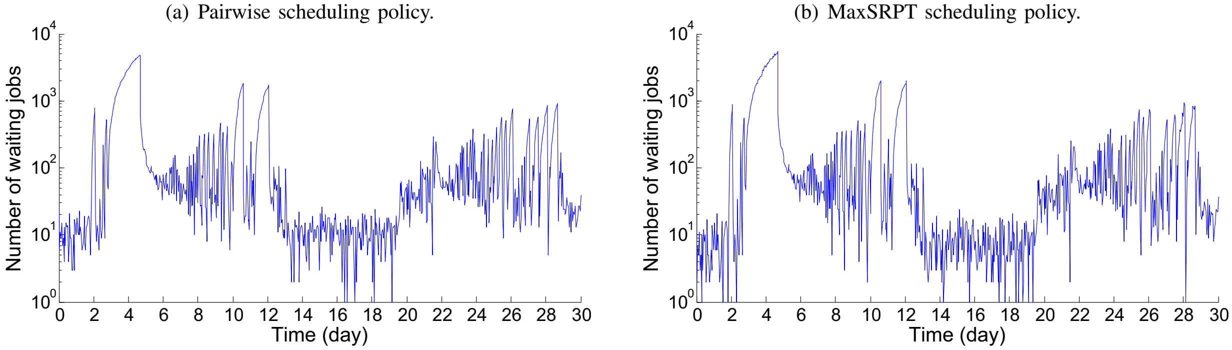
VII. CONCLUSION

This paper focuses on a joint scheduling optimization in MapReduce, where map and shuffle phases can be overlapped and be conducted in parallel. The scheduling objective is to minimize the average job makespan. The key challenge is that the map and shuffle phases cannot be fully parallelized due to their dependency relationship: the shuffle phase may



(a) Pairwise scheduling policy.

(b) MaxSRPT scheduling policy.



(c) Group-based scheduling policy.

(d) Online group-based scheduling policy.

Fig. 7. Online performance evaluation with respect to the number of waiting jobs.

wait to transfer the data emitted by the map phase. To avoid I/O underutilization, jobs that can form a strong pair should be pairwisely executed. Several offline and online scheduling policies are proposed to execute jobs in a pairwise manner. Scheduling optimalities are discussed under several scenarios. Finally, real data-driven experiments validate the efficiency and effectiveness of the proposed scheduling policies.

REFERENCES

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

[2] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz, "The case for evaluating mapreduce performance using workload suites," in *IEEE MASCOTS 2011*, pp. 390–399.

[3] M. Zaharia, A. Konwinski, A. D. Joseph, R. H. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments." in *USENIX OSDI 2008*, pp. 29–42.

[4] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals," in *ACM/USENIX Middleware 2011*, pp. 165–186.

[5] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *ACM EuroSys 2010*, pp. 265–278.

[6] A. Verma, L. Cherkasova, and R. H. Campbell, "Aria: automatic resource inference and allocation for mapreduce environments," in *ACM ICAC 2011*, pp. 235–244.

[7] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo, "Automated profiling and resource management of pig programs for meeting service level objectives," in *AMC ICAC 2012*, pp. 53–62.

[8] 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 *ACM/USENIX Middleware 2010*, pp. 1–20.

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

[10] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri." in *USENIX OSDI 2010*, pp. 24–33.

[11] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: optimizing mapreduce on heterogeneous clusters," *ACM SIGARCH Computer Architecture News*, vol. 40, no. 1, pp. 61–74, 2012.

[12] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *ACM SOSP 2009*, pp. 261–276.

[13] G. Ananthanarayanan, C. Douglas, R. Ramakrishnan, S. Rao, I. Stoica, I. Y. L. Dont, and B. Us, "True elasticity in multi-tenant clusters through amoeba," in *ACM SoCC 2012*, pp. 1–7.

[14] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella, "Multi-resource packing for cluster schedulers," in *ACM SIGCOMM 2014*, pp. 455–466.

[15] Q. Zhang, M. F. Zhani, Y. Yang, R. Boutaba, and B. Wong, "Prism: Fine-grained resource-aware scheduling for mapreduce," *IEEE Transactions on Cloud Computing*, vol. 3, no. 2, pp. 182–194, 2015.

[16] F. Ahmad, S. T. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Shufflewatcher: Shuffle-aware scheduling in multi-tenant mapreduce clusters," in *USENIX ATC 2014*, pp. 1–12.

[17] A. Verma, B. Cho, N. Zea, I. Gupta, and R. H. Campbell, "Breaking the mapreduce stage barrier," *Journal of Cluster Computing*, vol. 16, no. 1, pp. 191–206, 2013.

[18] M. Lin, L. Zhang, A. Wierman, and J. Tan, "Joint optimization of overlapping phases in mapreduce," *Performance Evaluation*, vol. 70, no. 10, pp. 720–735, 2013.

[19] J. Li, J. Wu, and X. Yang, "Optimizing mapreduce based on locality of kv pairs and overlap between shuffle and local reduce," in *IEEE ICPP 2015*, pp. 1–10.

[20] J. Wang, M. Qiu, B. Guo, and Z. Zong, "Phase-reconfigurable shuffle optimization for hadoop mapreduce," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2015.

[21] J. Wilkes, "More Google cluster data," Google research blog, Nov. 2011, posted at http://googleresearch.blogspot.com/2011/11/.

[22] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format + schema," Google Inc., Mountain View, CA, USA, Technical Report, Nov. 2011, revised 2012.03.20. Posted at http://code.google.com/p/googleclusterdata/wiki/TraceVersion2.