MROrder: Flexible Job Ordering Optimization
for Online MapReduce Workloads

Shanjiang Tang, Bu-Sung Lee, and Bingsheng He

School of Computer Engineering, Nanyang Technological University
{stang5,ebslee,bshe}@ntu.edu.sg

Abstract. MapReduce has become a widely used computing model for large-scale data processing in clusters and data centers. A MapReduce workload generally contains multiple jobs. Due to the general execution constraints that map tasks are executed before reduce tasks, different job execution orders in a MapReduce workload can have significantly different performance and system utilization. This paper proposes a prototype system called *MROrder* to dynamically optimize the job order for online MapReduce workloads. Moreover, MROrder is designed to be flexible for different optimization metrics, e.g., *makespan* and *total completion time*. The experimental results show that MROrder is able to improve the system performance by up to 31% for makespan and 176% for total completion time.

1 Introduction

MapReduce [1] is a popular computing paradigm for large-scale data intensive processing. A map-reduce job computation generally contains two phases: 1) a map phase, consisting of many map tasks, and 2) a reduce phase, consisting of many reduce tasks. Apache Hadoop, an open source framework of MapReduce, has been widely deployed on large clusters consisting of thousands of machines by companies such as Facebook, Amazon, and Yahoo. Generally, MapReduce and Hadoop are used to support batch processing for multiple large jobs (i.e., MapReduce workloads). Despite many research efforts have been devoted to improve the performance of a single MapReduce job (e.g., [1,2]), there is relatively little attention that has been paid to the system performance of MapReduce workloads. Therefore, this paper attempts to improve the system performance of MapReduce workloads.

The job execution order in a MapReduce workload is important for the system performance. To show the importance of job ordering, Figure 1 gives an example illustrating that the performance can differ by nearly 100% for two varied job submission orders for a batch of jobs. However, the job ordering optimization for MapReduce workloads is 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 [3].
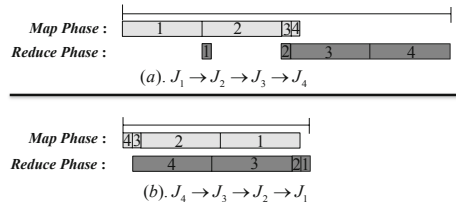
Fig. 1. Performance comparison for a batch of jobs under different job submission orders

The job ordering optimization for MapReduce workloads is important as well as 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 [3].

In this paper, we propose a prototype system *MROrder*[1] that can perform job ordering automatically for arriving jobs queued in Hadoop FIFO buffer. There are two core components for *MROrder*, namely, *policy module* and *ordering engine*. The policy module decides when and how to perform job ordering. The ordering engine, consisting of two approaches (i.e., simulation-based ordering approach and algorithm-based ordering approach), gives the job ordering. MROrder is designed to be flexible for different performance metrics, such as *makespan* and *total completion time*.

We evaluate our MROrder system using both synthetic workloads. Both makespan and total completion time are considered. Experimental results show that there is about 11–31% performance improvement based on MROrder system, depending on the characteristic of testbed workloads. Moreover, for synthetic Facebook workloads which contain lots of small-size jobs, the MROrder can improve the performance of the total completion time up to 176%.

2   Related Work

The batch job ordering optimization has been extensively researched in HPC literature [4]. In those studies, parallel tasks can be classified into three types: *rigid* (the number of processors to execute the task is fixed a priori), *moldable* (the number of processors to execute the task is not fixed but determined before the execution) and *malleable* (the number of processors for a task may change during the execution) [5]. In contrast, the *malleable task* is the most popular and widely studied. Its has been proved to be $\mathcal{NP}$-hard for *makespan* optimization [4], and a number of approximation and heuristic algorithms (e.g., [5,10]) were proposed. Meanwhile, there are some bi-criteria optimization algorithms proposed for optimizing *makespan* and *total completion time* simultaneously, such as [6].

---

[1] MROrder is open source and available at http://sourceforge.net/projects/mrorder/

The previous optimization works in HPC are implicitly targeted at the single-stage parallelism. In contrast, MapReduce is an interleaved parallel and sequential computation model [7]. It is close to the two-stage hybrid flow shop (HFS) [8]. Specifically, when each job contains only one map task and one reduce task, the MapReduce job ordering problem turns to be a two-stage HFS. The *makespan* optimization for two-stage HFS is strongly $\mathcal{NP}$-hard when at least one stage contains multiple processors [11]. There has been a large body of approximation and heuristic algorithms (e.g., [12,13]) for it. Besides, for HFS, there are also works (e.g., [15,14]) targeted at the bi-criteria optimization of both *makespan* and *total completion time*.

However, a MapReduce job runs multiple map/reduce tasks concurrently in each phase, which is different from the traditional HFS that allows only at most one task to be processed at a time. The MapReduce is more similar to the two-stage Hybrid flow shop with multiprocessor tasks (HFSMT) [16,17], which allows a task at each stage to be processed on multiple processors simultaneously. However, there is a strict requirement for HFSMT that a task at each stage can only be scheduled to execute only when there are enough idle processors for the task [17]. In contrast, the number of running map/reduce tasks for a MapReduce job is dynamically scaling up and down at runtime by allocating the tasks with available map/reduce slots.

In summary, this paper has taken into account all of these similarities to HFS as well as differences for MapReduce jobs. The most related work to us for MapReduce are [3,18]. Moseley et al. [3] presented a 12-approximation algorithm for the offline workloads of minimizing the *total flow time*, which is the sum of the time between the arrival and the completion of each job. Verma et al. [18] proposed two algorithms for *makespan* optimization of offline jobs. One is a greedy algorithm based on Johnson's Rule. The other one is a heuristic algorithm called BalancedPool. They evaluated their strength experimentally. In contrast, our work considers both *makespan* and *total completion time* optimization for *online* recurring MapReduce workloads, where jobs arrive over time and perform recurring computations in different time windows. Particularly, previous study showed that 75% of queries from Microsoft are recurring workloads [24]. MROrder is designed to optimize the performance for such scenarios.

3   Definition and Performance Metrics

Variable Definition. In an Hadoop cluster, let $\mathcal{M}$ denote the map phase and $\mathcal{R}$ denote the reduce phase. Let its slot configuration be $\mathcal{S} = \{\mathcal{S}^{\mathcal{M}}, \mathcal{S}^{\mathcal{R}}\}$, where $\mathcal{S}^{\mathcal{M}}$ denotes the set of map slots and $\mathcal{S}^{\mathcal{R}}$ denotes the set of reduce slots. Therefore, the *map task capacity* is $|\mathcal{S}^{\mathcal{M}}|$ and the *reduce task capacity* is $|\mathcal{S}^{\mathcal{R}}|$. For a batch of online jobs $J = \{J_1, J_2, ..., J_n\}$, let t_i^a denote the arriving time. Let $t_i^{\mathcal{M}}$ be the average processing time of a map task and $t_i^{\mathcal{R}}$ be the average processing time of a reduce task for each job J_i. Moreover, the set of its map tasks is denoted as $J_i^{\mathcal{M}}$ and the set of its reduce tasks is denoted as $J_i^{\mathcal{R}}$. Then the number of map tasks for J_i is $|J_i^{\mathcal{M}}|$ and the number of reduce tasks is $|J_i^{\mathcal{R}}|$.

Performance Metrics. There are several classical performance metrics for job ordering optimization, e.g., *makespan*, *total completion time* and *total flow time*. Let c_i denote

the *completion time* of the job J_i. The *makespan* for the whole jobs J is defined as the *maximum* completion time of any job, i.e., $C_{max} = \max\{c_i\}$. The *total completion time (TCT)* for the whole jobs J is defined as $C_{tct} = \sum c_i$.

Problem Definition. Our goal is to minimize *makespan* and *total completion time*. That is, how to order the *online* arriving jobs automatically such that the makespan (or the total completion time) for all the jobs is minimized?

4   MROrder System

This section describes the design and implementation of the *MROrder* system.

4.1   System Overview

Figure 2 presents the overall design architecture for *MROrder* system. It gives the job order for arriving jobs, which can be submitted by a user or from other softwares such as Pig, Hive, Mahout, Oozie, etc. Particularly, the jobs are submitted in an *ad hoc* manner from users. We do not have assumption for the arrival order as well as arrival rate of jobs. There is a *JobDispatchingQueue* for queueing arriving jobs before submitting them to the MapReduce cluster. The *MROrder* job ordering manager handles the job ordering for arriving jobs queued in *JobDispatchingQueue* automatically. For each MapReduce job, the *MROrder* system needs to know the following information, i.e., the number of map (or reduce) tasks, the average time for each map (or reduce) task, and its arriving time. There are two key components for *MROrder* job ordering manager, namely, *Policy Module* and *Ordering Engine*. The *policy module* determines when and how to perform job ordering for MapReduce jobs. Once a policy command is issued, the *ordering engine* then deals with the job ordering work automatically. The specific description for each component is detailed in the following sections.
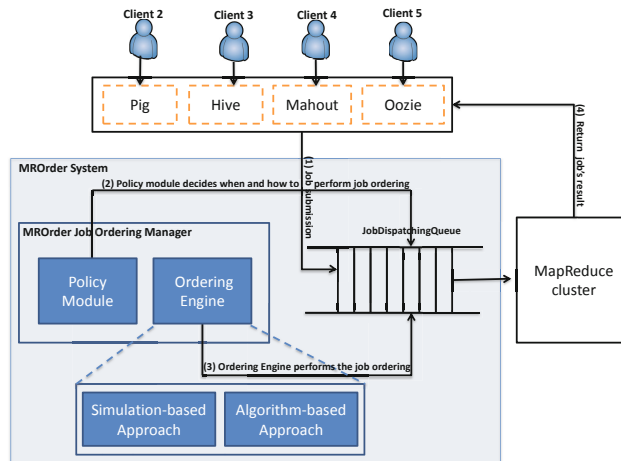

Fig. 2. The overall architecture for *MROrder* system

4.2 Policy Module

The policy module is invoked when there are arriving jobs queued in the *JobDispatchingQueue*, pending to be dispatched to the MapReduce cluster. It determines a good job ordering strategy to optimize target performance metrics (e.g., *makespan* or *total completion time*). The strategy is a combination of the choice of job ordering approach, the policy for the number of jobs for ordering and time policy (when to perform job ordering). It chooses the job ordering approaches (e.g., simulation-based approach, algorithm-based approach) based on their accuracy and efficiency characteristics (Section 4.3). Particularly, since simulation-based job ordering is a brute-force method, it can provide an optimal result but its efficiency is quite low, indicating that it is suitable for a small number of jobs. In contrast, the algorithm-based job ordering approach is efficient but it can only provide a sub-optimal result, which is suitable for a large number of jobs. Furthermore, the policy for the number of ordering jobs (PNJ) and time policy (TP) are correlated. We need to consider them together. We have the following two solutions:

PNJ-Dominated Solution. The user sets a threshold (n_0) for the number of jobs required to perform ordering. The ordering engine is triggered automatically when the number of arriving jobs reaches that threshold ($n \geqslant n_0$). The *TP* completely depends on the *PNJ*. It can be dynamically determined and computed by subtracting the latest-round job ordering time (or the starting time) by the current time.

TP-dominated Solution. Given a time interval Δt, the ordering engine is invoked at the time $t = \Delta t + t'$, where t' is the latest-round time when the ordering engine was activated (or the starting time). The number of jobs n is thus equivalent to the number of arriving jobs during this time interval. The TP-dominated solution is shown in Algorithm 1.

Algorithm 1. *TP-dominated Solution with Fixed Time Interval (TP-FTI)*

---

1. Assume that MapReduce cluster start at the time $t^{curr} = 0$. For each arriving job J_i, it will be first queued in the *JobDispatchingQueue*. There is a boolean attribute $orderflag_i$ for each J_i. It is initialized to be $orderflag_i = false$ by default.
2. The MROrder job ordering manager waits for a time interval Δt until the current time $t^{curr} = t^{curr} + \Delta t$. The policy module checks the arriving jobs queued in the *JobDispatchingQueue* to filter out sub-set J_A, where $J_A = \{J_i | (J_i \in J) \wedge (t_i^a \leqslant t^{curr}) \wedge (orderflag_i = false)\}$. Thus the number of jobs at this job ordering round is $|J_A|$.
3. The job ordering engine is triggered by the policy module. It does job ordering and marks $orderflag_i = true$ for jobs in J_A.
4. The MROrder system dispatches those jobs J_i with $orderflag_i = true$ in the *JobDispatchingQueue* and goes back to step 2.

Given $\Delta t = 60$ sec configured by the user, for example, the *MROrder* job ordering engine is activated every 60 secs, ordering the arriving jobs queued in the *JobDispatchingQueue* and dispatching them into MapReduce cluster. The value of Δt has a big

impact on the whole performance. Too small value of Δt can make the MROrder job ordering engine work so frequently that there may be a very few jobs available (e.g., 0, 1 or 2 jobs at each job ordering round) in the *JobDispatchingQueue* at each job ordering round, losing the effect of job ordering. However, too large value of Δt will make *JobDispatchingQueue* hold lots of jobs without distributing it to the MapReduce cluster, causing MapReduce cluster keep idle without running jobs and in turn have a adverse effect on the performance. Moreover, even we have a fine configuration for Δt, it is still inflexible and not adapted to the job arrival rate. We further propose an adaptive *TP* solution to solve this problem, as shown in Algorithm 2.

Algorithm 2. *TP-dominated Solution with Adaptive Time Interval (TP-ATI)*

---

1. Let MapReduce cluster start at the time $t^{curr} = 0$. For each arriving job J_i, it will be first queued in the *JobDispatchingQueue*. There is a boolean attribute $order flag_i$ for each J_i. It is initialized to be $order flag_i = false$ by default. Initially, let $t^{wait} = \Delta t$.
2. The MROrder job ordering manager waits for a time interval t^{wait} until the current time $t^{curr} = t^{curr} + t^{wait}$. The policy module checks the arriving jobs queued in the *JobDispatchingQueue* to filter out sub-set J_A, where $J_A = \{J_i | (J_i \in J) \wedge (t_i^a \leqslant t^{curr}) \wedge (order flag_i = false)\}$. Thus the number of jobs at this job ordering round is $|J_A|$.
3. The job ordering engine is triggered by the policy module. It does job ordering and marks $order flag_i = true$ for jobs in J_A.
4. The MROrder system dispatches those jobs J_i with $order flag_i = true$ in the *JobDispatchingQueue*.
5. The policy module updates t^{wait} as follows: $t^{wait} = \max\left\{\Delta t, T_A\right\}$, where $T_A = \max_{1 \leqslant k \leqslant |J_A|}\left\{\sum_{i=1}^{k} \frac{|J_i^{\mathcal{M}}| \cdot t_i^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|} + \sum_{i=k}^{|J_A|} \frac{|J_i^{\mathcal{R}}| \cdot t_i^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|}\right\}$.
6. Go back to Step 2.

The rationale for the adaptive waiting time adjustment based on the algorithm *TP-ATI* is that, user provides a relatively small threshold Δt for waiting time. The policy module adjusts it dynamically according to the estimated running time T_A of those workloads J_A that have been distributed to MapReduce cluster at the previous dispatching round. The MROrder tries to queue as many jobs as possible in the *JobDispatchingQueue* at each job ordering round while keeping the MapReduce cluster busy.

4.3 Ordering Engine

The ordering engine (OE) is triggered according to the policies in the policy module. The *MROrder* system provides two types of job ordering approaches, i.e., *simulation-based ordering approach* and *algorithm-based ordering approach*. The policy module is responsible for selecting the suitable ordering engine dynamically based on the number of jobs at each job ordering round. The basic idea is that the simulation-based ordering approach is chosen when there are a small number of jobs (e.g., 7 jobs), considering that it can produce an optimal result but is time-consuming. The algorithm-based ordering approach is selected for a large number of jobs.

Simulation-Based Ordering Approach (SIM). To enable simulation-based job ordering, we developed a Hadoop simulator named *HSim*. It is a tailored simulator aiming to evaluate the performance of varied job orders with a file input consisting of jobs information each with five arguments: job's ID, the number of map tasks, the number of reduce tasks, the average running time of a map task, the average running time of a reduce task. We build our simulation-based ordering approach based on *HSim*. It is a brute-force method that can enumerate all possible job orders to explore the optimal job order for a given performance metric (e.g., *makespan*, *total completion time*). Note that there are $n!$ possible job orders for n jobs. For example, there are $9! = 362880$ possible job orders for $n = 9$ jobs, which however takes 97.179 sec (refer to Table 2) for enumerating all job orders. It indicates that the simulation-based ordering approach is only feasible for a small number of jobs in practice. Moreover, instead of searching the whole space of all job orders, one might consider the Monte Calo method combined with HSim for suboptimal (rough) results by searching the partial space statistically for a large number of jobs (e.g., 50 jobs, 100 jobs). However, we argue that it is still not meaningful for a large number of jobs in practice. For example, assume that we want to control the maximum execution time of simulation not exceeding 97.179 sec (i.e., our sample space of job orders is 362880). When it comes to 20 jobs, it can only cover $\frac{362880}{20!=2432902008176640000} = 1.49 \times 10^{-13}$, which is very tiny and unmeaningful. Therefore, there is a need to explore an efficient solution for a large number of jobs in the following subsection.

Algorithm-Based Ordering Approach (ALG). We develop an algorithm-based ordering approach to deal with the job ordering for MapReduce workloads with a large number of jobs. It contains some job ordering greedy algorithms for different performance metrics. Particularly, we incorporate a greedy algorithm *MK* based on *Johnson's Rule* [9], as shown in Algorithm 3 for *makespan* optimization. It is an optimal and efficient $O(n \log n)$ job ordering algorithm for the makespan optimization for the two-stage flow shop with

---

Algorithm 3. Greedy algorithm based on *Johnson's Rule* (MK)

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:

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

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:

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

   b). Sort 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., $\phi_1 : J' = \{\{J_A\}, \{J_B\}\}$.

one processor per stage. The details of Johnson's rule is 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. $J \backslash J_A$). 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 optimized job order is obtained by appending the sorted set J_B to the end of sorted set J_A. Moreover, we also include a greedy algorithm *TCT* for the total completion time optimization, as shown in Algorithm 4, based on *shortest processing time first*. In comparison to the simulated-based ordering approach, the algorithm-based ordering approach is much more efficient, but it can only produce the sub-optimal result. Moreover, to support user's job ordering algorithms, *MROrder* system also provides a user interface in the algorithm-based ordering approach. Therefore, based on our *MROrder* system, user can extend the algorithm-based ordering approach for other's performance metrics.

Algorithm 4. Greedy algorithm based on *Shortest Processing Time First* (TCT)

---

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

$$T_i = T_i^{\mathcal{M}} + T_i^{\mathcal{R}} = \frac{|J_i^{\mathcal{M}}|}{|\mathcal{S}^{\mathcal{M}}|} \cdot t_i^{\mathcal{M}} + \frac{|J_i^{\mathcal{R}}|}{|\mathcal{S}^{\mathcal{R}}|} \cdot t_i^{\mathcal{R}}.$$

2. Order all jobs in J from left to right by non-decreasing T_i.

4.4   Implementation

We have developed a prototype of the *MROrder* system. The prototype implements all components of the *MROrder* job ordering manager. The policy module provides users with all policy solutions mentioned above for choices and adopts *TP-ATI* by default. Several user's arguments are provided, including the optimization targets (e.g., *makespan*, *total completion time*), the threshold for waiting-time interval as well as the maximum number of jobs allowed at each job ordering round. The *MROrder* system automates the corresponding job ordering policy in runtime based on user's argument configuration. Moreover, our prototype adopted our simulator *HSim* as the computing component of the MapReduce cluster to simulate the computation process of online MapReduce batch jobs. The current prototype primarily aims to study various automated policy solutions for online workloads under different performance metrics. It remains as ongoing work to incorporate it into Hadoop framework for practical use.

Data Skew. In our MROrder system, we assume that the sizes and processing time of all data blocks are the same, i.e., there is no data skew among data blocks. For the case of data skew, user can use the model provided by [26] to diminish it.

Overhead. The overhead of MROrder mainly comes from the ordering engine to perform job ordering. The detailed results are given in Section 5.3. Generally, SIM takes longer time than ALG, but it provides better performance result. Thus, there is a trade-off between the performance result and overhead for the dynamic choice of job ordering approach.

5   Experimental Evaluation

In this section, we evaluate MROrder prototype and its associated policies. The detailed evaluation method is that: first, we discuss and compare the effectiveness of proposed policy solutions (e.g., TP-FTI, TP-ATI). Then we evaluate and discuss the suitable value of threshold (of the number of jobs) as the condition for switching of job ordering approach. Third, we evaluate the performance for MROrder with regard to makespan as well as total completion time. Finally, we evaluate the accuracy of our simulator *Hsim* adopted by MROrder experimentally.

5.1   Workloads

Our experiment consists of two types of synthetic workloads. One is the synthetic Facebook workload, generated based on [19,20]. Specifically, the number of map/reduce tasks as well as the arriving time for each job are based the input/output data sizes of workloads provided by [19]. We estimate the running time of map and reduce tasks per job based on the map and reduce durations in Figure 1 of [20]. More precisely, we follow the LogNormal distribution [21] 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 [22]. It contains lots of small-size jobs (more than 58% in the number of jobs) [20]. We use it primarily to evaluate the total completion time for MROrder system.

Our second workload is a testbed workload. In contrast to synthetic Facebook workload, most of its jobs are large-size. The makespan is seriously affected primarily by the positions of large-size jobs. We use it mainly to evaluate the makespan for MROrder system.

5.2   Evaluation and Analysis of Policy Solutions

Recall that in the policy module of MROrder system, we provided several policy solutions to determine when and how to perform job ordering dynamically. Table 1 illustrates the comparison results of two policy solutions TP-FTI and TP-ATI for their suitable threshold Δt and the corresponding performance improvement of total completion time under varied sizes of synthetic Facebook workloads. Particularly, we evaluate different Δt from 10 sec, 20 sec, 30 sec till to 400 sec. We can observe that, (1). the suitable value of Δt for TP-FTI, TP-ATI is $230 \sim 350$ sec, and $10 \sim 30$ sec, respectively. It indicates that the threshold for the fixed-time interval method TP-FTI should be large, whereas it should be small for the adaptive method TP-ATI, relying on its adaptive mechanism to change the waiting time interval between two successive job ordering dynamically; (2). Under the suitable value of Δt, we note that the performance improvement of TP-ATI is much better than that of TP-FTI. This is because the TP-ATI is smarter than TP-FTI. Therefore, we take TP-ATI as the policy solution for the policy module in the following experiments.

Table 1. The comparison results of two different policy solutions for their suitable threshold Δt and the corresponding *performance improvement of total completion time* (PITCT) under varied sizes of synthetic Facebook workloads. The PITCT is a normalized ratio of performance improvement with MROrder to the unoptimized one.

JobNum	TP-FTI		TP-ATI	
	$\Delta t(sec)$	**PITCT** (%)	Δt (sec)	**PITCT** (%)
50	230	41.91	10	44.31
100	230	14.81	10	28.30
150	230	9.28	30	14.13
200	230	6.98	30	10.83
250	310	24.08	30	123.74
300	350	19.49	30	186.29
350	350	13.11	20	89.7
400	350	9.81	20	56.91

5.3 Switching Threshold for the Number of Jobs for Job Ordering Approach

In our MROrder prototype, we provide two types of job ordering approaches, namely, *SIM* and *ALG*. There is a tradeoff between the accuracy and overhead (i.e., the erased time it takes.) for these two ordering approaches (See Section 4.3 for details).

Table 2. Performance and overhead comparison of ALG versus SIM

| JobNum	Makespan for ALG (sec)	Makespan for SIM (sec)	Total Completion Time for ALG (sec)	Total Completion Time for SIM (sec)	Erased Time for ALG (sec)	Erased Time for SIM (sec)
1	45	45	45	45	0.001	0.002
2	170	170	240	240	0.001	0.003
3	200	198	456	453	0.003	0.003
4	338	324	799	796	0.003	0.003
5	399	394	1342	1274	0.003	0.028
6	399	396	1450	1363	0.003	0.123
7	440	437	1766	1736	0.003	0.952
8	475	471	2107	2050	0.003	9.305
9	573	564	2728	2596	0.004	97.179

Table 2 presents the comparison results under different numbers of jobs (e.g., 1-9) from our testbed workload. It consists of three parts. Column 2 and 3 give the results for makespan. Column 4 and 5 show the results for total completion time. Column 6 and 7 give the overheads for ALG and SIM ordering engines. We can observe that, (1). The results based on SIM ordering engine are better (more minimal) than that of ALGs for both makespan and total completion time. This is because SIM is a brute-force method that searches all possible job orders to get an optimal one, whereas ALGs are greedy algorithms that can only produce suboptimal results. (2). The results produced by ALG are close to SIM results, especially for makespan produced by algorithm *MK*. (3). The erased time (i.e., the overhead) consumed by ALG is very small and does not grow much

as the number of jobs increases. However, the erased time for SIM grows exponentially as the number of jobs increases, especially when the number of jobs equals to 9 (e.g., 97.179 sec). It is because ALGs are $n \log n$ algorithms, whereas SIM is an $n!$ brute-force method. Based on the experimental erased time and performance results, we set the threshold for the number of jobs to be 7 as a threshold for the dynamical choice of job ordering engines.

5.4 Performance Evaluation of MROrder System

We evaluate the performance of MROrder system by considering two metrics (i.e., makespan, and total completion time) and two kinds of workloads (e.g., Facebook workloads and testbed workloads). In our MROrder, we take TP-ATI for the policy module with Δt of 10 sec.
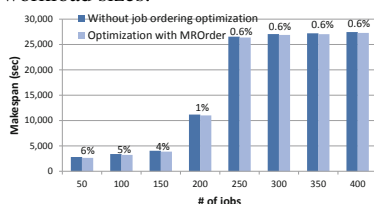
Figure 3 presents optimized performance results based on MROrder system, under varied sizes of online workloads. Specifically, the results for testbed workloads are shown in Figure 3 (a) and Figure 3 (b). The results for synthetic Facebook workloads are shown in Figure 3 (c) and Figure 3 (d). There is about $11\% - 31\%$ makespan improvement for testbed workloads in Figure 3 (a), whereas there is only 3% for Facebook workloads on average in Figure 3 (c). It is because that the makespan is affected primarily by the position of large-size jobs. The testbed workloads contain lots of large-size jobs. In contrast, the Facebook workloads consist of a large number of small-size jobs. On the other hand, for total completion time, Figure 3 (d) illustrates that the maximum performance improvement can be up to 176% for synthetic Facebook workloads. In contrast, there is a maximum of 24% performance improvement for total completion time of testbed workloads. The reason is that the total completion time is primarily
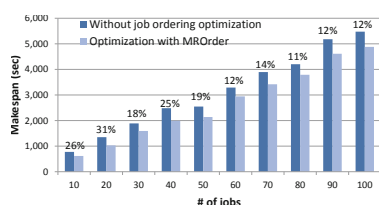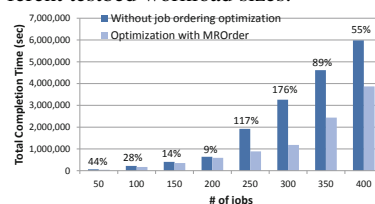

(a) Makespan under different testbed workload sizes.
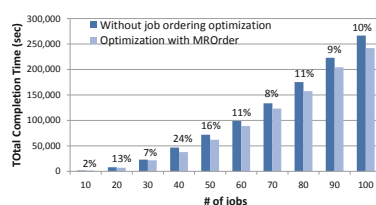
(b) Total completion time under different testbed workload sizes.

(c) Makespan under different synthetic Facebook workload sizes.

(d) Total completion time under different synthetic Facebook workload sizes.

Fig. 3. The optimized performance results for MROrder system under different sizes of testbed workloads and synthetic Facebook workloads

dominated by the positions of small-size jobs. The total completion time might be poor when there are lots of small-size jobs in a workload, e.g., Facebook workload.

5.5   Accuracy Evaluation for Hsim

We validate the accuracy of our *Hsim* by comparing the simulation results with the experimental results of a MapReduce workload. We generate our MapReduce workload by using three representative applications, i.e., **wordcount** application (computes the occurrence frequency of each word in a document), **sort** application (sorts the data in the input files in a dictionary order) and **grep** application (finds the matches of a regex in the input files). We take Wikipedia article history dataset[2] of 10GB, as application input data. We ran experiments in Amazon's Elastic Compute Cloud(EC2) [23]. Our EC2 Hadoop cluster consists of 20 nodes each belonging to a "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 4 virtual cores with 2 EC2 compute units each [23]. We configure 3 map and 1 reduce slots per slave node.

We consider the makespan as well as total completion time for all possible job orders of the MapReduce workload. Figure 4 (a) and Figure 4 (b) present the results for all $4! = 24$ job orders of a batch of 4 jobs. We note that the simulated results of both makespan and total completion time are very close (errors within 8%) to the experimental results, which validates the accuracy of our *Hsim*.
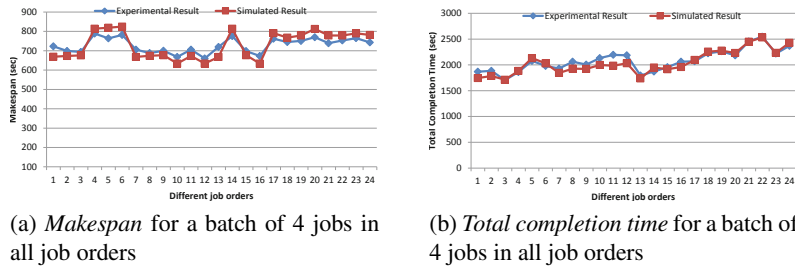

(a) *Makespan* for a batch of 4 jobs in all job orders

(b) *Total completion time* for a batch of 4 jobs in all job orders

Fig. 4. Simulated results versus experimental results for a MapReduce workload

6   Conclusion and Future Work

This paper proposed a prototype system named MROrder to perform job ordering optimization automatically for online MapReduce workloads. Several policy solutions were presented and evaluated to dynamically determine when and how to do job ordering. The MROrder system is designed to be flexible for different optimization metrics. It has implemented several algorithms to support the job ordering optimization for makespan and total completion time. It also provides an interface for users to add their job ordering algorithms into MROrder for optimization of other performance metrics.

---

[2] http://dumps.wikimedia.org/enwiki/

We are integrating MROrder into Hadoop framework. Moreover, our prototype for MapReduce only supports FIFO scheduling. In future, we will consider other schedulers such as Fair Scheduler [20], and heterogeneous environments such as [25].

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. OSDI (2004)
2. HowManyMapsAndReduces, http://wiki.apache.org/hadoop/HowManyMapsAndReduces
3. Moseley, B., Dasgupta, A., Kumar, R., Sarl, T.: On scheduling in map-reduce and flow-shops. SPAA, 289–298 (2011)
4. Leung, J.Y.T.: Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Chapman and Hall/CRC, 25-5-25-18 (2004)
5. Dutot, P.F., Mounie, G., Trystram, D.: Scheduling parallel tasks approximation algorithms. In: Leung, J.T. (ed.) Handbook of Scheduling: Algorithms, Models, and Performance Analysis, Chapman Hall, CRC Press (2004)
6. Dutot, P., Eyraud, L., Mounie, G., Trystram, D.: Bi-criteria Algorithm for Scheduling Jobs on Cluster Platforms. SPAA, pp. 125–132 (2004)
7. Howard, K., Siddharth, S., Sergei, V.: A model of computation for MapReduce. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 938–948 (2010)
8. Gupta, J.N.D.: Two stage hybrid flowshop scheduling problem. Journal of Operational Research Society 364, 359C–364C (1988)
9. Johnson, S.M.: Optimal two- and three-stage production schedules with setup times included. Naval Res Logist Q 1, 61–68 (1954)
10. Sanders, P., Speck, J.: Efficient Parallel Scheduling of Malleable Tasks. IPDPS, pp. 1156–1166 (2011)
11. Gupta, J.N.D., Hariri, A.M.A., Potts, C.N.: Scheduling a two-stage hybrid flow shop with parallel machines at the first stage. Annals Of Operations Research 69, 171–191 (1997)
12. Kyparisis, G.J., Koulamas, C.: A note on makespan minimization in two-stage flexible flow shops with uniform machines. European Journal of Operational Research 175, 1321–1327 (2006)
13. Hejazi, S.R., Saghafian, S.: Flowshop-scheduling problems with makespan criterion: a review. International Journal of Production Research 43, 2895–2929 (2005)
14. Gupta, J.N.D., Hennig, K., Werner, F.: Local search heuristics for two-stage flow shop problems with secondary criterion. Journal Computers and Operations Research 29, 123–149 (2002)
15. Rajendran, C.: Two-Stage Flowshop Scheduling Problem with Bicriteria. Journal of the Operational Research Society 43, 871–884 (1992)
16. Oğuz, C., Ercan, M.F., Cheng, T.C.E., Fung, Y.F.: Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop. European Journal of Operational Research 149, 390–403 (2003)

17. Oğuz, C., Ercan, M.F.: Scheduling multiprocessor tasks in a two-stage flow-shop environment. In: Proceedings of the 21st International Conference on Computers and Industrial Engineering, pp. 269–272 (1997)
18. Verma, A., Cherkasova, L., Campbell, R.: Two Sides of a Coin: Optimizing the Schedule of MapReduce Jobs to Minimize Their Makespan and Improve Cluster Performance. In: MASCOTS (2012)
19. Chen, Y.P., Ganapathi, A., Griffith, R., Katz, R.: The Case for Evaluating MapReduce Performance Using Workload Suites. In: MASCOTS (2011)
20. Zaharia, M., Borthakur, D., Sarma, J.: Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. In: EuroSys, pp. 265–278 (2010)
21. LogNormal Distribution,
    http://en.wikipedia.org/wiki/Log-normal_distribution
22. Verma, A., Cherkasova, L., Kumar, V.S., Campbell, R.H.: Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle. In: NOMS (2012)
23. Amazon EC2, http://aws.amazon.com/ec2
24. He, B.S., Yang, M., Guo, Z., Chen, R.S.: Comet: batched stream processing for data intensive distributed computing. In: SOCC, pp. 63–74 (2010)
25. Tan, Y.S., Lee, B.S., Campbell, R.H., He, B.S.: A Map-Reduce Based Framework for Heterogeneous Processing Element Cluster Environments. In: CCGrid (May 2012)
26. Ibrahim, S., Jin, H., Lu, L., He, B.S., Wu, S.: Adaptive I/O Scheduling for MapReduce in Virtualized Environment. ICPP 2011, 335–344 (2011)