Optimization techniques on job scheduling and resource allocation for MapReduce system

Tang, Shanjiang

2015

Tang, S. (2015). Optimization techniques on job scheduling and resource allocation for MapReduce system. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/64312>

<https://doi.org/10.32657/10356/64312>

NANYANG TECHNOLOGICAL UNIVERSITY

**Optimization Techniques on Job Scheduling and Resource
Allocation for MapReduce System**

Tang Shanjiang

Supervisor: Dr. Lee Bu-Sung

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in fulfillment of the requirement for the degree of Doctor of Philosophy

May, 2015

Abstract

MapReduce has become a popular high performance computing paradigm for large-scale data processing. Hadoop, an open source implementation of MapReduce, has been widely deployed in large clusters containing thousands of machines by companies such as Yahoo! and Facebook to support batch processing for large jobs submitted from multiple users (i.e., MapReduce workloads). However, there are certainly a lot of room to improve the performance and fairness of Hadoop. In this thesis, we focus on optimization techniques on job scheduling and resource allocation to improve the performance and fairness of Hadoop system.

First, we focus on the performance optimization for MapReduce workloads under FIFO scheduler without changing the source code of Hadoop by using job re-ordering approach. We consider two different kinds of production workloads, i.e., *offline* MapReduce workloads and *online* MapReduce workloads. The performance metrics used are makespan and total completion time. For *offline* workloads, we propose several job ordering algorithms. Based on the offline approaches, we further propose a prototype system called *MROrder* to optimize the performance for *online* MapReduce workloads. The experimental results show that our job ordering methods can significantly improve the performance of Hadoop for both offline and online workloads.

Second, instead of keeping the default static MapReduce resource allocation model where the number of map slots and reduce slots are pre-configured and not fungible¹, we relax the model constrain to allow slots to be reallocated to either map or reduce tasks depending on their needs through modifying the source code of Hadoop. A dynamic fair resource allocation and scheduling system called DynamicMR is proposed and implemented in Hadoop. It can improve the performance of MapReduce workloads while ensuring the fairness without any information about MapReduce jobs. The experimental results validate the effectiveness of our DynamicMR. Moreover, it can also be applied for FIFO scheduler.

¹Not fungible: we refer to the resource allocation constrain that map slots can only be allocated to map tasks, and reduce slots can only be used by reduce tasks

Finally, besides the optimization for Hadoop MRv1, we also optimize the fair resource allocation for YARN (i.e., Hadoop MRv2). Specifically, we consider pay-as-you-use computing (e.g., cloud computing) and find that the traditional fair policy is *not* suitable for such computing system. To address this, we propose a Long-Term Resource Fairness (LTRF) and implement it in YARN by developing LTYARN, a long-term YARN fair scheduler. Our experimental results show that it leads to a better resource fairness than existing fair scheduler.

Thus, in this thesis, we have addressed some of the optimization problems on job scheduling and resource allocation for MapReduce system under different scenarios. We have proposed new algorithms and frameworks to improve the performance and fairness for Hadoop system. The proposed algorithms and frameworks will be options for users who want to optimize the performance of their MapReduce workloads or ensure fairness, according to their needs and conditions.

Acknowledgments

First, I would like to express my sincere gratitude to my supervisor, Assoc. Prof. Bu Sung Lee, for his patience, invaluable advice and suggestions, and encouragement for my research work and the writing up of this thesis.

I would also like to express my thankfulness and whole-hearted gratitude to Asst. Prof. Bingsheng He for his patience, invaluable advice, suggestions, and comments for my research work.

I would like to express my sincere appreciation to my friends and colleagues in Parallel and Distributed Computing Centre (PDCC), especially Jin Jiangming, Sivadon Chaisiri, Alan Tan Yu Shyang, Irene Ng-Goh Siew Lai, Chonho Lee, and Changbing Chen, for their helpful discussions, kind assistance, and supports.

Last but not least, I am deeply grateful to my wife, my parents, my brothers and my sisters, for their everlasting support and endless love. This thesis is dedicated to them.

Contents

Abstract	i
Acknowledgments	iii
List of Figures	vii
List of Tables	ix
List of Notations	x
1 Introduction	1
1.1 Background for MapReduce and Hadoop	1
1.2 Motivations	2
1.2.1 Poor Resource Utilization Problem in Hadoop	3
1.2.2 Unfairness Problem for Hadoop Fair Scheduler in Pay-as-you-use Computing System	4
1.3 Objectives	5
1.4 Contributions	6
1.5 Organization of the Thesis	8
2 Literature Review	9
2.1 Performance Optimization for MapReduce	9
2.1.1 Resource Allocation and Job Scheduling Optimization Technique	9
2.1.2 Data Access and Sharing Optimization Techniques for MapReduce	13
2.1.3 Other Optimization Techniques	16
2.2 Fairness Optimization for MapReduce	17
2.2.1 Fairness Definitions and Metrics	17
2.2.2 Fair Scheduling Policies, Algorithms and Systems	18
2.3 Novelty of This Thesis	19
3 Job Ordering Optimization for MapReduce Workloads	21
3.1 Introduction	21
3.2 Motivation	22

3.3	Performance Optimization for Offline MapReduce Workloads	23
3.3.1	Problem Formulation and Performance Model	23
3.3.2	Makespan Optimization	27
3.3.3	Bi-criteria Optimization for Makespan and Total Completion Time .	29
3.3.4	Analysis of Job Ordering Algorithms	30
3.3.5	Practical Usage Consideration	31
3.3.6	Evaluation	32
3.4	Performance Optimization for Online MapReduce Workloads	38
3.4.1	Problem Definition	38
3.4.2	MROrder System	38
3.4.3	Evaluation	44
3.5	Summary	49
4	Dynamic Slot Allocation Optimization for MapReduce Cluster	51
4.1	Introduction	51
4.2	DynamicMR	52
4.2.1	System Overview	52
4.2.2	Dynamic Hadoop Slot Allocation (DHSA)	54
4.2.3	Speculative Execution Performance Balancing (SEPB)	64
4.2.4	Slot PreScheduling	67
4.2.5	DynamicMR Implementation	73
4.2.6	Dicussion	75
4.3	Evaluation	76
4.3.1	Experimental Setup	77
4.3.2	Performance Evaluation for DHSA	77
4.3.3	Speculative Execution Control for Performance	82
4.3.4	Data Locality Improvement Evaluation for Slot PreScheduling . . .	83
4.3.5	Performance Improvement Evaluation for DynamicMR	85
4.3.6	Performance Comparison With YARN	86
4.4	Comparison between DynamicMR and Job Ordering Optimization	89
4.5	Summary	90

5	LTyARN: A Long-Term YARN Fair Scheduler for Pay-as-you-go Computing	91
	Systems	91
5.1	Introduction	91
5.2	Allocation Properties and Fairness Definition	92
5.2.1	Resource Allocation Properties	92
5.2.2	Fairness Definition	93
5.3	MemoryLess Resource Fairness (MLRF)	94
5.4	Long-Term Resource Fairness(LTRF)	96
5.4.1	Property Analysis for LTRF	99
5.5	LTyARN: A Long-Term YARN Fair Scheduler	101
5.5.1	Long-Term Max-Min Fairness	101
5.5.2	Design and Implementation of LTyARN	107
5.6	Experimental Evaluation	110
5.6.1	LTRF Resource Allocation Flow	112
5.6.2	Fairness Results	113
5.6.3	Performance Results	116
5.6.4	Adaptive Task Quantum Policy Evaluation	117
5.7	Summary	118
6	Conclusion and Future Work	120
6.1	Conclusion	120
6.2	Future Works	123
	References	128
A	Theorem and Lemma in Chapter 3	139
A.1	Proof of Lemma 3.2	139
A.2	Proof of Lemma 3.3	140
A.3	Proof of Theorem 3.1	142
A.4	Proof of Theorem 3.2	143
A.5	Proof of Theorem 3.3	144

List of Figures

1.1	The architecture overview of Hadoop.	2
1.2	An illustrated example for the poor utilization inefficiency problem of Hadoop system under FIFO scheduler.	4
1.3	The overview of our project objectives.	6
3.1	Performance comparison for a batch of jobs under different job orders	22
3.2	MapReduce tasks execution flow for the simplified case.	25
3.3	MapReduce tasks execution flow for an example of four jobs $J_1, J(2), J(3), J(4)$ under the generalized case.	27
3.4	Normalized experimental results for testbed workloads under different job submission orders	33
3.5	Normalized simulation results for synthetic Facebook workloads under different job submission orders	35
3.6	The normalized comparison result of simulated makespan, lower bound makespan and upper bound makespan for optimal job order produced by <i>MK-JR</i>	36
3.7	The performance impact of map/reduce task execution time variation for different variation percentages.	37
3.8	The overall architecture for <i>MROrder</i> system.	39
3.9	The optimized performance results for MROrder system under different sizes of testbed workloads and synthetic Facebook workloads.	47
3.10	Simulated results versus experimental results for a MapReduce workload.	48
4.1	Overview of DynamicMR System.	53
4.2	Example of the fairness-based slot allocation flow for PI-DHSA.	56
4.3	Example of the fairness-based slot allocation flow for PD-DHSA.	59
4.4	The slot allocation flow for each pool under PD-DHSA.	62
4.5	The computation policy for the <i>totalNumOfPendingMapTasks</i> and <i>totalNumOfPendingReduceTasks</i>	67

4.6	An example of unbalanced workload distribution of running map tasks in a Hadoop cluster.	69
4.7	The scenarios for using Slot PreScheduling.	71
4.8	The overview design and implementation of Slot PreScheduling and its combination with PD-DHSA.	72
4.9	The execution flow for the two DHSAs.	78
4.10	The performance improvement by DHSA under various slot configuration for MapReduce workloads.	80
4.11	The performance results with different percentages of map (or reduce) slots borrowed.	81
4.12	The performance results with SEPB.	82
4.13	The data locality improvement for Slot PreScheduling.	84
4.14	The performance improvement under Slot PreScheduling.	85
4.15	The performance improvement with our DynamicMR system for MapReduce workloads.	85
4.16	The performance improvement with our DynamicMR system for MapReduce workloads under Hadoop FIFO scheduler.	86
4.17	The comparison results between YARN and DynamicMR for MapReduce workloads.	87
4.18	The network throughput comparison between YARN and DynamicMR for data shuffling over time.	88
5.1	The long-term max-min fairness models for <i>LT</i> YARN.	104
5.2	Overview of <i>LT</i> YARN.	107
5.3	The adaptive task quantum policy for YARN.	109
5.4	Overview of detailed fairness resource allocation flow for LTRF.	112
5.5	Comparison of fairness results over time for each workload under MLRF and LTRF in YARN.	115
5.6	The compared performance results for Static Partitioning, MLRF and LTRF.	116
5.7	The adaptive task quantum results for LTRF in one hour.	117
6.1	The demand of memory resource for a task is changing over time.	124
6.2	Heterogeneous resource demands for tasks from google traces[4].	125

List of Tables

1.1	Allocation results based on the <i>memoryless</i> max-min fairness policy.	5
3.1	List of key Notation.	25
3.2	The job information for Purdue MapReduce Benchmarks.	33
3.3	The job size distribution at Facebook	34
3.4	The comparison results of two different policy solutions under varied sizes of synthetic Facebook workloads	45
3.5	Performance and overhead comparison of ALG versus SIM.	46
4.1	Benefit and Cost Comparison between Slot PreScheduling and Delay Scheduler. .	73
4.2	The key functions in the DynamicMR.	74
4.3	The user configurable arguments for DynamicMR.	74
4.4	Benefit and cost comparison for allocation components regarding each metric. . .	76
4.5	The job information for Purdue MapReduce benchmarks and data sets.	77
5.1	A comparison example of <i>MLRF</i> and <i>LTRF</i>	97
5.2	List of properties for MLRF and LTRF.	101
5.3	Job types and sizes for each bin in our synthetic Facebook workloads.	111

List of Notations

$\mathcal{M}$	the <i>map</i> phase of MapReduce
$\mathcal{R}$	the <i>reduce</i> phase of MapReduce
J_i	the i^{th} MapReduce job
$ J_i^{\mathcal{M}} $	the number of map tasks for the i^{th} MapReduce job
$ J_i^{\mathcal{R}} $	the number of reduce tasks for the i^{th} MapReduce job
$ t_i^{\mathcal{M}} $	the average execution time for the map tasks of the i^{th} MapReduce job
$ t_i^{\mathcal{R}} $	the average execution time for the reduce tasks of the i^{th} MapReduce job
$\mathcal{S}^{\mathcal{M}}$	the number of map slots in the MapReduce cluster
$\mathcal{S}^{\mathcal{R}}$	the number of reduce slots in the MapReduce cluster
C_{max}	the makespan for a batch of jobs $J_1, J_2 \dots$
$\hat{C}_{max}$	the upper bound makespan for a batch of jobs $J_1, J_2 \dots$ under a job order
$\check{C}_{max}$	the lower bound makespan for a batch of jobs $J_1, J_2 \dots$ under a job order
$\hat{C}_{max}^*$	the upper bound makespan for a batch of jobs $J_1, J_2 \dots$ for all job orders
$\check{C}_{max}^*$	the lower bound makespan for a batch of jobs $J_1, J_2 \dots$ for all job orders
C_{tct}	the total completion time for a batch of jobs $J_1, J_2 \dots$
ϕ	a job submission order for a batch of jobs $J_1, J_2 \dots$
$T_i^{\mathcal{M}}$	the total processing time of map tasks of the i^{th} MapReduce job
$T_i^{\mathcal{R}}$	the total processing time of reduce tasks of the i^{th} MapReduce job
t_i^a	the arriving time for the i^{th} MapReduce job
Pec_M	the percentage of map slots borrowed by reduce tasks in DynamicMR
Pec_R	the percentage of reduce slots borrowed by map tasks in DynamicMR
$f_i(t)$	the <i>accumulated</i> resources for the i^{th} client at time t .
$g_i(t)$	the current allocated resources for the i^{th} client at time t
$d_i(t)$	the current demand for the i^{th} client at time t
$S_i(t)$	the current resource share for the i^{th} client at time t
$\beta_i(t)$	the fairness degree for the i^{th} client at time t
$\Psi(t)$	the <i>sharing benefit degree</i> under a fair policy
$\Omega(t)$	the <i>sharing loss degree</i> under a fair policy

$\tau_i(t)$	the set of tasks from the i^{th} application that are allocated with resources at the time t
$\theta_{i,j}$	the resource demand (e.g., the size of vcore or memory) for the j^{th} task of the i^{th} application
$\delta_{i,j}$	the resource weight for the j^{th} task of the i^{th} application
$\alpha_{i,j}(t)$	the <i>assumed execution time</i> for the j^{th} task of the i^{th} application at time t
Λ_i	the i^{th} applications
$n_i^p(t)$	the number of pending (i.e., runnable) tasks for the application Λ_i at time t
ω_i	the shared weight for the i^{th} application
L	the time length of a computation round in Semi-Long-Term Max-Min Fairness Model
t^s	the start time of the current computation round in Semi-Long-Term Max-Min Fairness Model
$t_{i,j}^s$	the start time for the j^{th} task of the application Λ_i
$t_{i,j}^c$	the finished time for the j^{th} task of the application Λ_i

Chapter 1

Introduction

This chapter first presents some background on MapReduce and Hadoop. We then show the motivations and objectives of our work by identifying the shortcomings of current Hadoop system. Finally, we highlight the contribution of our work and give the organization of the thesis.

1.1 Background for MapReduce and Hadoop

MapReduce is a popular programming model proposed by Google[29]. Now it has been a de facto standard for large scale data processing on a dedicated computing environment like a cluster[29], grid[53] or cloud[78]. Hadoop[5] is an open-source java implementation of MapReduce. It runs on top of a distributed file system called HDFS, which splits an input data into multiple blocks of fixed size (typically 64MB) and replicates each data block several times across computing nodes. Users can submit MapReduce jobs to the Hadoop cluster. The Hadoop system breaks each job into multiple map tasks and reduce tasks, with its map tasks computed before its reduce tasks. Each map task processes (i.e. scans and records) a data block and produces intermediate results in the form of key-value pairs. Generally, the number of map tasks for a job is related to input data size. There is one map task per data block. The execution time for a map task is determined by the data size of an input block. The reduce tasks consists of shuffle/sort/reduce phases. In the shuffle phase, the reduce tasks fetch the intermediate outputs from each map task. In the sort/reduce phase, the reduce tasks sort intermediate data and then aggregate the intermediate values for each key to produce the final output. The number of reduce tasks for a job depends

on the intermediate map outputs. We can empirically set the number of reduce tasks for a job to be $0.95 \times$ or $1.75 \times$ reduce tasks capacity (i.e., the total number of reduce slots)[7]. The MapReduce paradigm is very suitable for a wide range of data-intensive and compute-intensive applications whose data processing can be parallelized into multiple map tasks followed by a set of reduce tasks. The typical applications include distributed pattern-based searching, distributed sorting, web link-graph reversal, web access log stats, inverted index construction, document clustering and machine learning. However, it is also worth mentioning that MapReduce is not suitable for applications with strong DAG dependencies between tasks or data, such as workflow and graph processing.

There are several job schedulers used in Hadoop, i.e., *FIFO*, *Hadoop Fair Scheduler*[100] and *Capacity Scheduler*[2]. The job scheduling is performed by the jobTracker (master), which manages a set of taskTrackers (slaves). Each taskTracker has a fixed number of map slots and reduce slots, as shown in Figure 1.1. Map tasks can only use map slots and reduce tasks can only be allocated with reduce slots. Typically, there is one slot per CPU core to ease the CPU and memory management of slave nodes[100]. The taskTrackers report periodically to the jobTracker the number of free slots and the progress of the running tasks. The jobTracker allocates the free slots to the running jobs. For each job, the reduce slots can only be allocated to it when at least one of its map tasks finishes.

Figure 1.1: The architecture overview of Hadoop.

1.2 Motivations

Performance and fairness are two key metrics for Hadoop system. This section motivates the importance of our work by showing problems on these two aspects. It first concerns the

low resource utilization problem, resulting in the poor performance, under the current design and resource allocation policy of Hadoop. Two important performance metrics are considered, namely, *makespan* and *total completion time*. Generally, the makespan is referred to as the maximum completion time of all jobs (See formal definition in Section 3.3.1.1). It is used to measure the performance and utilization efficiency of a system. It considers the computation time of all jobs. In contrast, the total completion time is computed as the sum of completion time of all jobs, as the formal definition given in Section 3.3.1.1. It can be viewed as a generalized makespan with queuing time (i.e., waiting time) included, which is used to measure the satisfaction from a single job's perspective by dividing the total completion time by the number of jobs (i.e., average completion time). There are different meanings/purposes for these two performance metrics. That is, the makespan is evaluating the performance from a global system view, whereas the total completion time is measuring the performance from an individual user/job's view.

Next, it reviews the Hadoop Fair Scheduler in pay-as-you-use computing system and points out the unfairness problem.

1.2.1 Poor Resource Utilization Problem in Hadoop

The compute resources (e.g., CPU cores) of the Hadoop cluster are abstracted into map and reduce slots, which are basic compute units and statically configured by administrator in advance. A MapReduce job execution has two unique features: 1) the slot allocation constraint assumption that map slots can only be allocated to map tasks and reduce slots can only be allocated to reduce tasks, and 2) the general execution constraint that map tasks are executed before reduce tasks. Due to these features, the resources are not fully utilized.

Figure 1.2 shows an example of multiple jobs scheduled using Hadoop FIFO scheduler. Suppose that there are four jobs $J_1 \sim J_4$ with job submission order of $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$. Let's assume that the map/reduce slot configuration for the cluster is 8/4. Moreover, we assume that the numbers of map tasks for J_1, J_2, J_3, J_4 are 10, 4, 2, 24, and their numbers of reduce tasks are 2, 4, 4, 6, respectively. We see that, there are lots of idle map/reduce slots that cannot be fully utilized during the execution, resulting in the low resource utilization and poor performance for the Hadoop cluster. For example, all map slots are in idle status,

Figure 1.2: An illustrated example for the poor utilization inefficiency problem of Hadoop system under FIFO scheduler. The submission order is $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$. Let's suppose that the map/reduce slot configuration for the cluster is 8/4. Moreover, we assume that the numbers of map tasks for J_1, J_2, J_3, J_4 are 10, 4, 2, 24, and their numbers of reduce tasks are 2, 4, 4, 6, respectively. We see that lots of idle map/reduce slots are available during the execution, resulting in the low resource utilization and poor performance for the Hadoop cluster.

being wasted between timeslot 28 to 44 when reduce tasks are running. Thus, to improve the performance, we need to have some approaches to maximize the slot utilization by making slots as busy as possible whenever there are pending tasks, instead of keeping slots idle before the computation of MapReduce workload completes. Finally, it is worth mentioning that the poor utilization problem does also exist in Hadoop Fair Scheduler (HFS), due to the rigid execution order that map tasks run before reduce tasks. For example, reduce slots will be idle and wasted at the beginning of MapReduce jobs' computation under HFS.

1.2.2 Unfairness Problem for Hadoop Fair Scheduler in Pay-as-you-use Computing System

In current pay-as-you-use computing systems, including supercomputers, data centers (e.g., Amazon EC2), there are ten to thousands of clients concurrently running their high-performance computing applications at any time. Clients pay on the basis of their resource usage. As a service-level agreement (SLA) guarantee, resource providers should guarantee that the amount of service a client receive is proportional to her payment over time.

Moreover, (*weighted*) *max-min fairness*[10] is one of the most popular fair allocation policy and has been used in Hadoop Fair Scheduler (HFS). Unfortunately, we observe that the fair policy implemented in Hadoop is *memoryless*, i.e., allocating resources fairly at instant time without considering history information, making it failed to satisfy the SLA mentioned above.

Here is an example (See more details in Section 5.4 of Chapter 5) to demonstrate this point. Consider a shared computing system consisting of 100 resources (e.g., 100GB RAM) and two users A, B with equal share of 50GB each. As illustrated in Table 1.1, assume that the new requested demands at time t_1, t_2, t_3, t_4 for client A are 20, 40, 80, 60, and for client B are 100, 60, 50, 50, respectively. With memoryless max-min fairness policy, we see in Table 1.1 that, at t_1 , the total demand and allocation for A are both 20. It lends 30 unused resources to B and thus 80 allocations for B . The scenario is similar at t_2 . Next at t_3 and t_4 , the total demand for A becomes 80 and 90, bigger than its share of 50. However, it can only get 50 allocations based on the memoryless max-min fairness, being *unfair* for A since it makes the total allocations for A and B become $160 (= 20 + 40 + 50 + 50)$ and $240 (= 80 + 60 + 50 + 50)$ at time t_4 , respectively.

Therefore, it is necessary and important for us to make a new fairness policy for Hadoop so as to meet the SLA requirement of pay-as-you-use computing system.

	Client A					Client B				
	Demand		Allocation		Preempt	Demand		Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	20	20	20	20	-30	100	100	80	80	+30
t_2	40	40	40	60	-10	60	80	60	140	+10
t_3	80	80	50	110	0	50	70	50	190	0
t_4	60	90	50	160	0	50	70	50	240	0

Table 1.1: Allocation results based on the *memoryless* max-min fairness policy. *Total Demand* refers to the sum of the new demand and accumulated remaining demand in previous time.

1.3 Objectives

In this thesis, we focus on the optimization for Hadoop schedulers. Both Hadoop FIFO Scheduler and Hadoop Fair Scheduler are considered. Specifically, we optimize two metrics

for Hadoop, namely, *performance* and *fairness*. The main objectives for this project are summarized as follows:

- To provide algorithms for improving the performance of MapReduce workloads in the case that users cannot change or have no permission to update the Hadoop system.
- To maximize the performance of MapReduce workloads when we can update or modify the source code of the Hadoop system.
- To improve the performance while guaranteeing the fairness for Hadoop Fair Scheduler.
- To explore approaches that can achieve the economic fairness requirement for Hadoop in pay-as-you-use computing system.

1.4 Contributions

Figure 1.3: The overview of our project objectives.

Figure 1.3 outlines the whole optimization work of the thesis for Hadoop, summarized as follows:

- *Job ordering optimization for Hadoop FIFO scheduler:* We note that in Hadoop FIFO scheduler, different job submission orders can have different performance for a batch of jobs. Based on this finding, we propose algorithms to optimize the performance for a batch of jobs through re-ordering their job submission order. Two kinds of performance metrics are taken into account, namely, makespan and total completion time. Moreover, two types of workloads are considered. The first workload is the *offline* MapReduce workloads where each of its jobs arrives at the same time. We propose two job ordering algorithms for this workload to optimize these two metrics respectively. Second, we consider the job ordering optimization for online MapReduce workloads where each job arrives over time. To enable the online optimization, we have developed a prototype called *MROrder* that combine the proposed offline job ordering algorithms to support the online job ordering optimization for different performance metrics.
- *MapReduce slot optimization for Hadoop clusters:* We have developed a slot optimization framework called *DynamicMR* for Hadoop Fair Scheduler, to improve the performance for MapReduce workloads while guaranteeing the fairness without any knowledge or assumption of the MapReduce jobs. It consists of three different optimization technique, i.e., Dynamic Hadoop Slot Allocation(DHSA), Speculative Execution Performance Balancing (SEPB), and Slot PreScheduling. We have integrated *DynamicMR* into Hadoop and the experiments demonstrate the effectiveness of our system.
- *Enabling economic fairness for YARN on pay-as-you-use computing systems.* We find that, the current fair policy of YARN is memoryless, making it failed to achieve the economic fairness for YARN users in a shared pay-as-you-use computing system. To address the problem, we propose a long-term resource fairness and show that it can it is suitable for pay-as-you-use computing. We have developed a fair scheduler called *LTYARN* for YARN and experimentally demonstrates the goodness of long-term resource fairness.

1.5 Organization of the Thesis

This thesis makes up of six chapters.

- Chapter 2 reviews some related work on the performance and fairness optimization of MapReduce including scheduling and resource allocation optimization, speculative task optimization, data locality optimizations; fairness definition, policies and algorithms.
- In Chapter 3, we propose job ordering optimization algorithms and system for the performance optimization of MapReduce workloads under Hadoop FIFO scheduler.
- A dynamic slot optimization system called DynamicMR is proposed in Chapter 4 to improve the performance for Hadoop Fair Scheduler while ensuring the fairness.
- Chapter 5 propose a new fair scheduler called LTYARN for YARN on pay-as-you-use computing system to achieve the economic fairness.
- In Chapter 6, we conclude the thesis and give out our future work.

Chapter 2

Literature Review

There are lots of research work focusing on job optimization in MapReduce[33]. In this chapter, we summarize the existing works that are related to ours from two aspects. First, we will describe some of the optimisation techniques developed for MapReduce. Next, we will review the related work on fairness for MapReduce.

2.1 Performance Optimization for MapReduce

In this section, we classify the related work on MapReduce performance optimization into three parts. The first part is about resource allocation and job scheduling optimization. The second part is regarding data access and sharing optimization. The third part belongs to other optimization techniques.

2.1.1 Resource Allocation and Job Scheduling Optimization Technique

1. *Job Ordering Optimization.* The batch job ordering problem has been extensively studied in high performance computing literature[71]. In those studies, parallel tasks are classified into three types: *rigid task* (the number of processors to execute the task is fixed a priori), *moldable task* (the number of processors to execute the task is not fixed but determined before the execution) and *malleable task* (the number of processors for a task may change during the execution)[41]. The *malleable task* is the most popular and widely studied. Its has been proved to be $\mathcal{NP}$ -hard for *makespan* optimization[71], and a number of approximation and heuristic job ordering algorithms (e.g., [20, 63, 41, 85, 38]) were proposed. Meanwhile, there are some bi-criteria op-

timization algorithms proposed for optimizing *makespan* and *total completion time* simultaneously, such as[35].

All of these works are 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[65]. It is close to the two-stage hybrid flow shop (2HFS) problem[49]. The *makespan* optimization for two-stage HFS is strongly $\mathcal{NP}$ -hard when at least one stage contains multiple processors[48]. There has been a large body of approximation and heuristic algorithms (e.g., [70, 55]) for two-stage HFS. Besides, for HFS, there are also works (e.g., [80, 50]) on the bi-criteria optimization for *makespan* and *total completion time*.

Moreover, 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)[77, 76], 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 when there are enough idle processors for the task[76]. In contrast, the number of running map/reduce tasks for a MapReduce job is dynamically scaled up and down at runtime by allocating the tasks with available map/reduce slots[73, 89, 93]. Moseley et al. [73] 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. [93] proposed an greedy algorithm based on Johnson's Rule for *makespan* optimization of offline jobs. They evaluated their strength experimentally. MROrder[89] considered 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.

2. *Data Locality Optimization*. Putting the task computation close to the node located with data has shown to be an efficient way to improve the performance (i.e., data locality). For MapReduce, there are *map-side* and *reduce-side* data locality. The

map-side data locality optimization considers moving the map tasks computation close to the input data blocks (e.g., [100, 78, 45, 46]). For example, when there are lots of small-size jobs in a environment, Delay Scheduler can improve the data locality by delaying the scheduling of map tasks whose data locality cannot be satisfied for a short period of time, at the expense of fairness[100]. Purlieus[78] classified the MapReduce jobs into three types, map-input heavy, map-and-reduce-input heavy and reduce-input heavy, and proposes data and virtual machine locality-awareness placement strategies accordingly in a cloud environment to improve the runtime performance. Guo et al. [45, 46] proposed a mathematical model and analyzed the relationship between system factors and data locality theoretically. Moreover, they also provided an algorithm for users to balance the tradeoff between fairness and data locality. In contrast, the *reduce-side data locality* optimization considers placing reduce tasks to the machines that are closest to the majority of the already generated intermediate data by map tasks, either by proposing some greedy algorithms (e.g., [78, 52, 51]) or by formulating the reduce task assignment problem as a stochastic optimization problem theoretically (e.g., [88, 87]).

3. *Straggler Mitigation Optimization*. The execution time of a MapReduce job is very sensitive to slow-running tasks (namely *straggler*)[97]. There are various reasons that cause stragglers, including faulty hardware and software mis-configuration[102]. Speculative execution is an important task scheduling strategy in MapReduce for dealing with straggler problem for a single job, including LATE[102], BASE[47], Mantri[17], MCP[24]. Longest Approximate Time to End (LATE)[102] is a speculative execution algorithm that focuses on heterogeneous environments by prioritizing tasks to speculate, selecting fast nodes to run on, and capping speculative tasks. Guo et al. [47] further improved the performance for LATE by proposing a Benefit Aware Speculative Execution (BASE) algorithm that can evaluate the potential benefit of speculative tasks and eliminate unnecessary runs. Mantri[17] provided a speculative execution strategy that focuses on saving cluster computing resource, i.e., task slots, by monitoring tasks and culling outliers based on their causes. Chen et al. [24]

proposed a new speculative execution algorithm called Maximum Cost Performance (MCP) to overcome the problems that affect the performance for previous speculative execution strategies, including data skew, task that starts asynchronously, and improper configuration of phase percentage. However, it is worth mentioning that all speculative executions mentioned above are not free. They come at the cost of cluster efficiency[97], which could have a negative impact for the performance of a batch of jobs.

4. *Load Balancing Optimization.* Keeping load balance is a non-trivial but challenging issue for MapReduce jobs. Kwon et al. [69] summarized four types of workload unbalanced causes (i.e. skews) during MapReduce job computation. In the Map phase, the expensive record (i.e., some records may need more CPU and memory to process than others) and heterogeneous map (i.e., each dataset of the map task may require different processing operations.) are two main contributors. In contrast, for the reduce phase, the workload unbalance problem is mainly from the uneven partitioning skew and expensiveness of key group. Some of the techniques proposed to address the load balancing problems are as follows:

One technique is *data pre-processing and sampling*. Kolb et al. [67] proposed two approaches (*BlockSplit* and *PairRange*) to handle data skew of MapReduce jobs in the context of entity resolution. Ramakrishnan et al. [81] focused on the problem of reduce keys with large loads. They achieved load balance by splitting the large-size reduce keys into multiple medium-size reduce keys, and then assigning medium-size keys to reduce tasks. Identifying such keys is via sampling and statistical modeling before the MapReduce job starts.

Another technique is *data repartitioning*. Gufler et al. [44] presented an adaptive load balancing strategy based on a proposed partitioning cost model for MapReduce. It is used to quantify the cost of the work assigned to reduce tasks. The global data distribution is approximated by computing local statistics in the map phase and aggregating them to produce global statistics. With such information, load balancing can then be achieved by assigning the data output from the map phase to reduce tasks adaptively.

SkewReduce[68] achieves load balancing by an optimizer utilizing user-defined cost functions to determine how best to partition the input data to minimize computational skew. SkewTune[69] mitigates both skew caused by an uneven distribution of data to operator partitions and skew due to some input records taking longer to process than others, with no extra user-supplied information. It is achieved by repartitioning the straggler's remaining unprocessed input data to one or more slots.

The third technique is *batching*. For example, Sailfish[82] proposes an abstraction technique called I-files that can aggregate intermediate data produced by map tasks and later consumed by reduce tasks, to improve the performance by batching disk I/O. Moreover, the similar technique of batching has also been adopted by Themis[83], a I/O efficient MapReduce system.

2.1.2 Data Access and Sharing Optimization Techniques for MapReduce

1. *Data and workload Sharing Approach.* For MapReduce, as a batch processing system, there are often lots of MapReduce jobs (or MapReduce queries) that share some parts of data or identical work. One optimization approach would be *data and workload sharing*. MRShare[75] is a sharing framework that reduces the redundant processing for batch jobs by sharing some portions of common data and workload, including data scan sharing, Map output sharing, Map Function sharing. It transforms a batch of MapReduce jobs into several groups and treat each group as a single job based on a cost model by solving an optimization problem regarding execution time minimization. To process a group, MRShare uses a tagging method to tag the sharing parts to save I/O computation. The experimental results show that it can significantly improve the performance of MapReduce jobs. Incoop[21] focuses on the incremental MapReduce workloads that often needs to recompute with small changes of input data. It can dynamically detect changes to the input and compute only the modified data only, to avoid recomputing the whole data. ReStore[36] is a system that manages the storage and reuse of intermediate results of the current workflow for

future workflow. Besides, there are several sharing work targeting at scan only, including[15,96,98].

2. *Data Access Approach.* Efficient data access to the input data stored in HDFS is a non-trivial step to improve the performance of MapReduce jobs. To achieve data access efficiency, we first need to have a data positioning mechanism that can find the desired data quickly. Moreover, instead of scanning all input data, accessing desired data only (i.e., avoiding unnecessary data) is also an efficient solution.

One technique is called *data indexing*, a classical method borrowed from DBMS domain. It creates an index for each input data block and allows users to search their desired data with the index, making data access be much faster than simply scanning. Hadoop++[31] modifies the HADFS system. It creates and injects an indexing information (called Trojan Indexes) into input splits and serves as a cover index for the data inside the split. Specifically, it creates a "Trojan" file by changing the traditional data set with a header information needed for indexing. Moreover, "Trojan Join" is also provided to locate the related data together by creating a new indexed file for join operation. HAIL (Hadoop Aggressive Indexing Library)[32] improves the upload pipelining of HDFS by creating different clustered indexes on each replica of block data while uploading data to HDFS. No additional MapReduce jobs are required to create those indexes. The decision on the indexes to create can either be done by a user through a configuration file or by a physical design algorithm. It typically improves both the upload times (even if index creation is included) and the MapReduce job execution times. It has a major advantage over Hadoop++ that the long upload and indexing times that has to be invested on previous systems are not required any more.

The second technique is called *Intentional data placement*. It improves the data access by intentionally putting the co-related data together so as to minimizing the network I/O overheads. Cohadoop[37] colocates and copartitions data on nodes intentionally, so that related data are stored on the same node. It modifies the existing

Hadoop file system with the additional parameter locator and groups the related files with the same locators on the same datanodes to satisfy the data locality.

The third technique is called *data layouts*. It organizes and stores the data in an easy-access manner for data queries. Llama[72] proposes a column-wise file format called *CFile* to achieve better performance than existing format in Hadoop. It partitions the data into vertical groups and sorts each group according to a selected column, which enables selective access to the columns used in the queries only and thus provides better performance. Cheetah[25] also stores the data in columnar format and applies different compression techniques for different types of values appropriately. It employs a PAX layout[16] at the block level so that each block contains the same set of rows and column layout is employed inside the block. Compared to Llama, it can avoid expensive network access due to its feature that all data belonging to a record are stored in the same block. RCFile[54] avoids the wasted processing by combining horizontal with vertical partitioning according to decompression of unnecessary data. It partitions the data into splits horizontally first, and then partitions each horizontal split vertically. It can avoid network access since columns belonging to the same row are located together on the same node. CIF[40] presents a column-oriented, binary storage format for Hadoop. It partitions each file into splits and store each split into a subdirectory, where the columns of the records of each split are stored in individual files, together with a file containing metadata about the schema. CIF outperforms RCFile and no changes to the core of Hadoop is needed.

3. *Caching and Pipelining Technique.* HaLoop[23] focuses on the performance improvement for iterative applications (e.g., page ranking). It is achieved by making the task scheduler loop-aware and by adding various caching mechanisms. The loop-aware task scheduling approach enables data reuse across iterations, by physically co-locating tasks that process the same data in different iterations. The caching for loop-invariant data can reduce the I/O cost for loading and shuffling in subsequent iterations. HaLoop uses three types of caches: the map task and reduce task input caches, as well as the reduce task output cache. In addition, to improve performance,

cached data are indexed. MapReduce Online[28] changes the built-in materialization mechanism for the output results of map tasks and reduce tasks, by providing support for pipelining of intermediate data from map tasks to reduce tasks. Particularly, the modifications include the data transfer between map tasks and reduce tasks within a job, and between reduce tasks and new map tasks across jobs. After changes, it enables Hadoop to perform online aggregation for an approximation of the final answers and continuous queries, which are not supported by default Hadoop.

2.1.3 Other Optimization Techniques

1. *Parameter Tuning Approach.* Hadoop is a flexible batch processing system consisting of more than 190 configuration parameters. The proper configuration of each argument is important for performance optimization of MapReduce jobs, which is however both cumbersome and unclear for the users to achieve[19]. Starfish[56] proposes a cost-based optimization approach for MapReduce program, determining the appropriate values for each configuration parameters of MapReduce jobs, e.g., the number of map and reduce tasks, the amount of memory per task. Inside Starfish, there is a profiler that collects statistics about the usage of resources, the execution time for each task and the size of the data. A what-if engine and a cost-based optimizer are adopted to search for suitable values for configuration arguments that result in good performance by using simulations and a model-based estimation method.
2. *Program Optimization.* Manimal[62] is a system for automatic analysis and optimization for data-intensive MapReduce programs. It improves the performance by exploiting the classical query optimization techniques (B+Trees for selections, column-store style techniques for projections, etc) existed in the traditional database. Hadoop-ToSQL[61] aims to utilize SQL's features for indexing, aggregation and grouping from MapReduce. It uses the static method to analyze the java code of a MapReduce query so as to completely or partially transform it to SQL. With such method, it can access only parts of data sets when indexes are available, instead of full scan. It can be applied for certain classes of MapReduce queries, e.g., a MapReduce function with no loops.

2.2 Fairness Optimization for MapReduce

In this part, we review related works on fairness. We first present the fairness definition and metrics in Section 2.2.1. Next, the fair scheduling policies, algorithms and systems are shown in 2.2.2.

2.2.1 Fairness Definitions and Metrics

In High Performance Computing(HPC), *Fairness* is an important criterion[34]. However, there is no unanimity in its definition. Sabin et al. [84] made a study on fairness metrics in parallel job scheduling. They classified the fairness into two broad types. The first type of fairness model is based on the social justice and is centered on the user's perception of fairness according to the expected order of service (e.g., arriving order). It is widely used in social and operational research studies. The second type of fairness is based on the fair partition of the resources or servers among multiple users. They proposed a concept of fair start time as a mean to measure social justice (i.e., belonging to the first type of fairness) for parallel job schedulers by defining as fair if no later arriving job delays an earlier arriving job. John et al. [74] gave a comparison on different fairness metrics on dispersion, fair start time, and queueing time. They observed that some of the metrics used to measure fairness for parallel job schedulers can imply unfairness where there is no discrimination (and vice versa). In view of it, they proposed a new approach called *net benefit* to measure fairness for parallel job schedulers. It is based on two principles: first, there is no need for jobs to have the same performance for the scheduler to be fair since they generally have different resource requirements and go into different queue/system states; second, the comparison of fairness can be made by considering how the schedulers favor/discriminate individual jobs. Zhao et al. [103] and Arabnejad et al. [18] considered fairness among multiple workflows. They defined fairness on the basis of *slowdown* that each workflow would experience, where the *slowdown* refers to the difference in the expected execution time for the same workflow when scheduled together with other workflows and when scheduled alone. Note that the former definitions of fairness are mainly based the 'performance' metrics such as start time, response time, waiting time and slowdown. When comes to the pay-as-you-use computing environment, our work shows that they are no longer suitable measure of fairness due

to the different user concerns in pay-as-you-go computing system(eg. cloud computing). Pay-as-you-use computing system is a service-oriented platform with SLA guarantee. That is, from service providers' perspective (e.g., Amazon, supercomputer operator), they do not care about what types of applications the clients are running (e.g., on AWS EC2, supercomputers) and the performance (i.e., the performance metrics for client's applications are not the main concerns for providers). Instead, they only need to guarantee the amount of resources allocated to each client over a period of time. In view of this, Tang et al. [90] proposed a Long-Term Resource Fairness (LTRF) based on this point in the shared pay-as-you-use computing system. It attempts to make sure that the total amount of resources each client obtained in a shared pay-as-you-use computing system is larger or at least the same as that in a non-shared partitioning system, according to her payment.

2.2.2 Fair Scheduling Policies, Algorithms and Systems

There are various kinds of fair policies in the traditional HPC and grid computing, including round-robin[34], proportional resource sharing[94], weighted fair queuing[30], and max-min fairness[10]. In comparison, max-min fairness is the most popular and widely used policy in many existing parallel and distributed systems such as Hadoop[97], YARN[91], Mesos[57], Choosy[43], Quincy[60]. Hadoop[97] partitions resources into slots and allocates them fairly across pools and jobs. In contrast, YARN[91] divides resources into containers (i.e., a set of various resources like memory, cpu) and tries to guarantee fairness between queues. Mesos[57] enables multiple diverse computing frameworks such as Hadoop, Spark sharing a single system. It proposes a distributed two-level scheduling mechanism called resource offers that decides how many resources to offer each framework and framework decides which resources to accept or which computation to run on them. Choosy[43] extends the max-min fairness by considering placement constraints. Quincy[60] is a fair scheduler for Dryad that achieves the fair scheduling of multiple jobs by formulating it as a min-cost flow problem. In addition to the single-resource fairness, there are some work focusing on multi-resource fairness, including Dominant Resource Fairness (DRF)[42] and its extensions[22, 66, 95, 79]. Ghodsi et al. [42] first generalized the max-min fairness from a single-resource fairness to multi-resource fairness by proposing DRF. They show that

DRF satisfies several important properties such as sharing incentive, strategy-proofness, envy-freeness and pareto efficiency. Bhattacharya et al. [22] extended DRF from a single-level to hierarchical levels by proposing a H-DRF scheduler. Kash et al. [66] considered a dynamic setting scenario for multi-resource allocation. They proposed a dynamic model of fair division and presented desirable axiomatic properties for dynamic resource allocation mechanism. Wang et al. [95] studied the multi-resource fairness in cloud computing systems with heterogeneous servers and proposed a DRFH scheduler by extending DRF. Parkes et al. [79] extended DRF in three directions. First, it shows that DRF generalizes to more expressive settings, and leverages a new technical framework to formally extend its guarantees. Second, it demonstrates that DRF performs poorly in terms of social welfare. We have showed that this is an unavoidable shortcoming that is shared by every mechanism that satisfies one of three basic properties. Lastly, it studies a realistic setting that involves indivisibilities. Note that all the previous algorithms are *memoryless*, i.e., instant fairness without considering historical information. Tang et al. [90] showed that there are three problems for them to be applied in pay-as-you-use computing systems and proposed a Long-Term Resource Fairness (LTRF) to fix it.

2.3 Novelty of This Thesis

In the present state-of-the-art, we propose the novel job scheduling and resource allocations techniques to optimize the performance and fairness for Hadoop system. In this thesis, different optimization algorithms are proposed under different scenarios. Specifically, we consider job ordering techniques for MapReduce workloads in the slot-based Hadoop system when users cannot change the system configuration. In view that this approach can only improve the utilization and performance at a certain degree, we further propose a new system by modifying the Hadoop source code to substantially improve the performance of MapReduce workloads. In addition to Hadoop MRv1, we also focus on optimization for YARN (i.e., Hadoop MRv2), viewing that its fair policy cannot support the pay-as-you-use computing system (e.g., cloud computing). We then propose a novel fair policy as well as a new YARN scheduler and show that it meets some desirable properties for pay-as-you-use

computing system. In conclusion, our work complements existing studies on performance and fairness for MapReduce system.

Chapter 3

Job Ordering Optimization for MapReduce Workloads

3.1 Introduction

In Hadoop MRv1, the resources are organized into slots, i.e., map slots and reduce slots. Given slots configured by the administrator, there are general execution constraints for MapReduce jobs that, 1) map tasks can only run in map slots and reduce tasks can only run in reduce slots, and 2) map tasks are executed before reduce tasks. Due to these, job execution order can have significantly impact on the performance for MapReduce jobs under the FIFO scheduler. It implies that Hadoop users can improve the performance for their workloads by optimizing the sequence in which jobs are submitted to the system.

In this chapter, we attempt to improve the performance of MapReduce workloads by adopting the job ordering optimization technique. Two important performance metrics are considered, namely, *makespan* (i.e., the maximum completion time of all jobs) and *total completion time* (i.e., the sum of completion time of all jobs). As discussed in Section 1.2, these two metrics are at different levels and have different meanings. That is, makespan is a performance metric for all jobs from a global view, whereas the total completion time can be used as the local performance metric for an individual user/job. In our study, we consider the optimization for two kinds of workloads, namely, offline workload (i.e., a batch of jobs each of which arrive at the same time) and online workloads (i.e., a batch of jobs that arrive at different time).

The rest of the chapter is organized as follows. Section 3.2 motivates the importance and effectiveness of job ordering optimization approaches. Section 3.3 proposes job ordering

optimization algorithms for *offline* MapReduce workloads. Section 3.4 considers *online* MapReduce workloads and proposes a flexible job ordering optimization system called MROrder. Finally, we summarize the chapter in Section 3.5.

3.2 Motivation

To appreciate the impact of job ordering optimization, we have performed an experiment with a testbed workload by enumerating all job submission orders. It consists of 4 MapReduce jobs from Table 3.2 in a Amazon EC2 Hadoop cluster configured with map/reduce slots of 57/19. Both *makespan* and *total completion time* (defined formally in Section 3.3.1.1) are considered. We test the results for all job submission orders of $4! = 24$.

(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.

Figure 3.1: Performance comparison for a batch of jobs under different job orders

Figure 3.1(a) shows the sorted results according to *makespan*, which is the time needed to complete all the jobs. We observe that there is a 20% (130 second) difference between

the best and worst job orderings. Depending on the characteristic of the workload, this difference can be larger. Figure 3.1(b) shows another set of jobs where the makespan can differ by nearly 100%. 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 can only run on map slots and reduce tasks can only use reduce slots, (iii). Both map slots and reduce slots are limited computing resources, configured by hadoop administrator in advance[73].

Moreover, in Figure 3.1(a), the curve running through the middle of the figure gives the *total completion time*, i.e. the sum of the completion times of all the jobs, of the different orderings. We see that there is a significant tradeoff between the makespan and total completion time. 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 job order that has the minimum makespan. It indicates that there is a need to optimize both makespan and total completion time together.

3.3 Performance Optimization for Offline MapReduce Workloads

3.3.1 Problem Formulation and Performance Model

3.3.1.1 Problem Formulation

A MapReduce *job* J_i 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 j^{th} map task and j^{th} reduce task, respectively. A MapReduce *workload* is a set of jobs $J = \{J_1, J_2, \dots, J_n\}$, 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.

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 $J_1, \dots, J_n$ is defined as $C_{max} = \max_{i \in [n]} \{c_i\}$. The *total completion time (TCT)* for the workload is defined as $C_{tct} = \sum_{i \in [n]} c_i$. Finally, Table 3.1 lists the key notations used in this chapter.

Based on the motivating example shown in Section 3.2, in this chapter, we consider two optimization problems, defined as follows:

Problem 1 Find an *ordering* ϕ to execute the jobs $J_1, \dots, J_n$ in a MapReduce workload such that C_{max} is minimized ?

Problem 2 Find an *ordering* ϕ to execute the jobs $J_1, \dots, J_n$ in a MapReduce workload that can optimize (minimize) C_{max} and C_{tct} simultaneously ?

Recall in Section 1.2 that both makespan and total completion time are important and meaningful in practice. Problem 1 focuses on the resource utilization (or overall execution time) optimization for all jobs. In contrast, Problem 2 considers the scenario that we want to optimize both of them.

3.3.1.2 Performance Model

In this section, we give a closed-form formula for makespan and total completion time when the number of map and reduce tasks are divisible by the number of map and reduce slots, and we also give an upper bound on the makespan when the values are not divisible. The latter is used in the following sections, where we describe the *MK-JR* algorithm and also prove its approximation ratio.

We first consider a simplified case where $|\mathcal{S}^M| = 1$ and $|\mathcal{S}^R| = 1$. It turns out to be a perfect two-machine flow-shop problem[93]. Figure 3.2 gives an example of an execution for this case. For each job J_i , let T_i^M be the total processing time of map tasks and T_i^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 $T_i^M = \frac{\sum_{j=1}^{|J_i^M|} t_{i,j}^M}{|\mathcal{S}^M|}$

Symbol	Description
$\mathcal{M}$	the <i>map</i> phase of MapReduce.
$\mathcal{R}$	the <i>reduce</i> phase of MapReduce.
J_i	the i^{th} MapReduce job.
$ J_i^{\mathcal{M}} $	the number of map tasks for the i^{th} MapReduce job.
$ J_i^{\mathcal{R}} $	the number of reduce tasks for the i^{th} MapReduce job.
$ t_i^{\mathcal{M}} $	the average execution time for the map tasks of the i^{th} MapReduce job.
$ t_i^{\mathcal{R}} $	the average execution time for the reduce tasks of the i^{th} MapReduce job.
$\mathcal{S}^{\mathcal{M}}$	the number of map slots in the MapReduce cluster.
$\mathcal{S}^{\mathcal{R}}$	the number of reduce slots in the MapReduce cluster.
C_{max}	the makespan for a batch of jobs $J_1, J_2, \dots$.
$\hat{C}_{max}$	the upper bound makespan for a batch of jobs $J_1, J_2, \dots$ under a job order.
$\check{C}_{max}$	the lower bound makespan for a batch of jobs $J_1, J_2, \dots$ under a job order.
$\hat{C}_{max}^*$	the upper bound makespan for a batch of jobs $J_1, J_2, \dots$ for all job orders.
$\check{C}_{max}^*$	the lower bound makespan for a batch of jobs $J_1, J_2, \dots$ for all job orders.
C_{tct}	the total completion time for a batch of jobs $J_1, J_2, \dots$.
ϕ	a job submission order for a batch of jobs $J_1, J_2, \dots$.
$T_i^{\mathcal{M}}$	the total processing time of map tasks of the i^{th} MapReduce job.
$T_i^{\mathcal{R}}$	the total processing time of reduce tasks of the i^{th} MapReduce job.
t_i^a	the arriving time for the i^{th} MapReduce job.

Table 3.1: List of key Notation.

Figure 3.2: MapReduce tasks execution flow for the simplified case.

and $T_i^{\mathcal{R}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|S^{\mathcal{R}}|}$. Based on the Johnson's work[64], the makespan and total completion time for the simplified case can therefore be calculated as follows:

$$X_k = \max\left\{\sum_{i=1}^k T_i^{\mathcal{M}} - \sum_{i=1}^{k-1} (T_i^{\mathcal{R}} + X_i), 0\right\}. \quad (3.1)$$

$$\sum_{i=1}^n X_i = \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k T_i^{\mathcal{M}} - \sum_{i=1}^{k-1} T_i^{\mathcal{R}} \right\}. \quad (3.2)$$

$$C_{max} = \sum_{i=1}^n (X_i + T_i^{\mathcal{R}}) = \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^n T_i^{\mathcal{R}} \right\}. \quad (3.3)$$

$$C_{tct} = \sum_{u=1}^n \sum_{i=1}^u (X_i + T_i^{\mathcal{R}}) = \sum_{u=1}^n \max_{1 \leq k \leq u} \left\{ \sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^u T_i^{\mathcal{R}} \right\}. \quad (3.4)$$

According to Johnson's work[64], there is a useful property about k which maximizes the quantity above for makespan as stated by the following lemma below,

Lemma 3.1 For C_{max} in Equation (3.3). if $C_{max} = \sum_{i=1}^{k_0} T_i^{\mathcal{M}} + \sum_{i=k_0}^n T_i^{\mathcal{R}}$, ($1 \leq k_0 \leq n$), there must be $X_{k'} = 0$ for all $k_0 < k' \leq n$; (2). Conversely, if $X_{k_0} > 0$ and $X_{k'} = 0$ for all $k_0 < k' \leq n$, there must be $C_{max} = \sum_{i=1}^{k_0} T_i^{\mathcal{M}} + \sum_{i=k_0}^n T_i^{\mathcal{R}}$, ($1 \leq k_0 \leq n$).

Lemma 3.1 simplifies Equation 3.3 for makespan. We will use it later for the proof of Lemma 3.2.

Next, we consider the general case where there is arbitrary number of map and reduce slots, as illustrated in Figure 3.3. In this case, it is difficult to give a closed-form formula for makespan and total completion time. Instead, we can compute 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 3.3.1.1 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 3.3.6. 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 3.3. 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 3.2 Given a job order ϕ , there is a upper bound *makespan* denoted by $\hat{C}_{max}$ (i.e. $C_{max} \leq \hat{C}_{max}$) for the generalized case as follows:

$$\hat{C}_{max} = \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k T_i^{\mathcal{M}} + \max_{1 \leq i \leq k} \{\hat{t}_i^{\mathcal{M}}\} + \sum_{i=k}^n T_i^{\mathcal{R}} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\} \right\}. \quad (3.5)$$

where $T_i^{\mathcal{M}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|S^{\mathcal{M}}|}$, $T_i^{\mathcal{R}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|S^{\mathcal{R}}|}$, $\hat{t}_i^{\mathcal{M}} = \max_{1 \leq j \leq |J_i^{\mathcal{M}}|} \hat{t}_{i,j}^{\mathcal{M}}$ and $\hat{t}_i^{\mathcal{R}} = \max_{1 \leq j \leq |J_i^{\mathcal{R}}|} \hat{t}_{i,j}^{\mathcal{R}}$.

The detailed proof of Lemma 3.2 is given in Appendix A.1.

Figure 3.3: MapReduce tasks execution flow for an example of four jobs $J_1, J(2), J(3), J(4)$ under the generalized case, where $|\mathcal{S}^{\mathcal{M}}| = 8, |\mathcal{S}^{\mathcal{R}}| = 4$. Assume that for J_1 there are 10 map tasks of 4 time units each, and 2 reduce tasks of 8 time units each. For J_2 , it has 4 map tasks of 2 time units each and 4 reduce tasks of 2 time units each. For J_3 , there are 2 map tasks of 6 time units each and 4 reduce tasks of 10 time units each. In contrast, for J_4 , it has 24 map tasks of 6 time units each and 6 reduce tasks of 8 time units each. The job submission order is $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$. Then the makespan $\hat{C}_{max}$ is equal to 44 and the total completion time $\hat{C}_{txt}$ is equal to $16 + 8 + 26 + 44 = 94$.

3.3.2 Makespan Optimization

Recall the simplified case described in the previous subsection, where the number of map and reduce tasks of all the jobs are 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*[64], 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. [93] first noted it and proposed an algorithm based on Johnson's rule. as we re-format in the following algorithm *MK_JR*, whose time complexity is $O(n \log n)$. To study the makespan result on it, in the rest part, we give a theoretical analysis based on our previous performance model in Section 3.3.1.2.

Theorem 3.1 *MK_JR* is an $(1 + \delta)$ -approximation algorithm for makespan optimization in the generalized case, where $\delta = \frac{\max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{M}}\} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\}}{\max_{1 \leq k \leq n} \{\sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^n T_i^{\mathcal{R}}\}}, (0 \leq \delta \leq 1)$.

Algorithm 1 Greedy algorithm based on *Johnson's Rule* (*MK-JR*)

Input:
 J : the MapReduce workload.

 $|\mathcal{S}^{\mathcal{M}}|$: the number of map slots.

 $|\mathcal{S}^{\mathcal{R}}|$: the 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 :

$$J_A = \{J_i | (J_i \in J) \wedge (T_i^{\mathcal{M}} \leq T_i^{\mathcal{R}})\},$$

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

- 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 Theorem 3.1, we need to compute an upper bound makespan and a lower bound makespan for all job orderings.

Lemma 3.3 Let $\check{\phi}^*$ denote the job order produced by *MK-JR*. Let $\hat{\phi}^*$ be the reverse 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 $\hat{C}_{max}^*$, for all job orders Φ . Particularly, $\check{C}_{max}^*$ is estimated with regard to $\check{\phi}^*$ by using Formula (3.3). $\hat{C}_{max}^*$ is estimated with regard to $\hat{\phi}^*$ with the Formula:

$$\hat{C}_{max}^* = \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k T_i^{\mathcal{M}} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{M}}\} + \sum_{i=k}^n T_i^{\mathcal{R}} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\} \right\}. \quad (3.6)$$

To prove Lemma 3.3, we need to find a worst-case job order for a batch of jobs under the simplified case such that its makespan, denoted as C_{max}^{wst} , is maximized. We find the following interesting relationship between the optimal job order and the worst-case job order, for the simplified case where the number of (map or reduce) tasks divides the number of (map or reduce) slots.

Theorem 3.2 Suppose ϕ is the optimal job order whose makespan is C_{max}^{opt} , 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_{max}^{wst} , can be obtained by simply reversing ϕ .

The proof of Theorem 3.2 is given in Appendix A.4. We now can prove Lemma 3.3 based on Theorem 3.2 and Lemma 3.2. The detailed proof of Lemma 3.3 is given in Appendix A.2.

Finally, the main Theorem 3.1 can be proved based on Lemma 3.2 and Lemma 3.3, as given in Appendix A.3.

In practice, 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.

3.3.3 Bi-criteria Optimization for Makespan and Total Completion Time

So far, we focus only on the optimization of makespan. Note that the total completion time that can be high, as illustrated in Figure 3.1 (a). Therefore, there is a need for bi-criteria optimization of 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 constraints[13]. However, *MK_JR* is not aware of varying job sizes. Indeed, the job order produced by *MK_JR* might have adverse effect on the total completion time if we follow *Johnson's Rule* strictly. For example, there can be a job J_i whose processing time (e.g. $T_i^M + T_i^R$) is very small but $T_i^M > T_i^R$. 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*.

In *MK_TCT_JR*, we first divide jobs set J into two sub-sets, 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[27]. Note that moving small-size jobs forward benefits the total completion time but hurts 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. Finally, the time complexity of *MK_TCT_JR* is $O(n \log n)$.

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

Input:

J : the MapReduce workload.

$|S^M|$: the number of map slots.

$|S^R|$: the 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:

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

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

$$J'_A = \{J_i | (J_i \in J) \wedge (T_i \leq T)\}, \quad J'_B = \{J_i | (J_i \in J) \wedge (T_i > T)\}$$

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., $\phi_2 : J' = \{\{J'_A\}, \{J'_B\}\}$.
-

3.3.4 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 to be removed from the MapReduce cluster. Therefore, an issue arises about

its influence on the optimized job orders produced by *MK_JR* and *MK_TCT_JR*. That is, is makespan (or total completion time) the *same* for the optimized job orders produced by *MK_JR* (or *MK_TCT_JR*) under different number of slave nodes?

In 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 3.3 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 depend on the number of map/reduce slots configured within a slave node.

The proof of Theorem 3.3 is shown in Appendix A.5. 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 3.3 does not mean the *optimal* job orders are *unchanged* under varied numbers of nodes.

For example, Let us consider a Hadoop cluster with 5 nodes, each configured with 2 map and 2 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 1 reduce slot. Let J_2 be defined as follows: Map stage duration is 11 and requires 8 map slots and reduce stage duration is 15 and requires 1 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 a node fails, there are only 4 nodes left with 8 map and 8 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.

3.3.5 Practical Usage Consideration

To use the proposed algorithms in practice, there are two key information about MapReduce jobs that we need to know in advance. First, the number of map tasks and reduce tasks for

each MapReduce job should be provided. We can deduce the number of map tasks based on the sizes of input data and data block. The number of reduce tasks can be obtained based on a setting guidance[7]. Second, we also need to get the execution time for map and reduce tasks. One way is based on historical information for those jobs (e.g., production jobs) that run periodically (e.g., in data warehouse). Another feasible method is to have a profiling run for MapReduce jobs with a small-size input data for obtaining the average execution time for map and reduce tasks.

Moreover, there are also some practical issues that are unpredictable, such as data locality, failed tasks and stragglers, that could have a negative impact on the performance of MapReduce jobs. In our current work, we assume that the cluster environment is fine and the impact of these issues is minor. It remains our future work to take these complex issues into account in our algorithms.

3.3.6 Evaluation

In this section, we evaluate our proposed algorithms using two kinds of workloads. First, we perform experiments with a testbed workload by running MapReduce jobs in Amazon's Elastic Compute Cloud (EC2)[1], as listed in Table 3.2. 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, 15 GB RAM and four 420GB 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. Third, we evaluate the tightness of proposed lower bound makespan as well as upper bound makespan with the Facebook workload experimentally.

3.3.6.1 Experiment Result with Testbed Workload

To reflect real workloads, we generate our testbed workloads by choosing 9 benchmarks arbitrarily from Purdue MapReduce Benchmarks Suite¹ and using their provided datasets.

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

Benchmark	# of map tasks	# of reduce tasks	Avg time for map tasks (sec)	Avg time for reduce tasks (sec)
Wordcount	160	100	22	11
Sort	320	200	9	24
Grep	480	120	9	11
Inverted-Index	640	100	32	23
Classification	160	120	6	13
Histogram-Movies	160	150	6	13
Histogram-Ratings	160	100	18	15
Sequence-Count	320	150	38	21
Tera-Sort	160	100	10	26

Table 3.2: The job information for Purdue MapReduce Benchmarks.

In the following experiment, we take the average execution time as input for our algorithms. Particularly, we validate that it is suitable for using average execution time in our algorithms by showing that the impact of this assumption is minimal in Section 3.3.6.4 experimentally. Table 3.2 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 $\tilde{t}_i^M$ and $\tilde{t}_i^R$ as shown in Table 3.2. Three different sizes of testbed workloads are generated, i.e., 10 jobs, 20 jobs, 30 jobs, with these 10 benchmarks.

(a) *Makespan speedup* under different number of jobs (b) *Total completion time speedup* under different number of jobsFigure 3.4: Normalized experimental results for testbed workloads under different job submission orders in a Hadoop cluster with $|\mathcal{S}^M| = 57$ and $|\mathcal{S}^R| = 19$.

Figure 3.4 presents the normalized performance results for testbed workloads under three different job orders, i.e., an unoptimized job order based on Theorem 3.2, the job order based on MK_JR and the job order based on MK_TCT_JR , in a Hadoop cluster, where

we configure 3 map slots and 1 reduce slot per slave node. Therefore, we have $|\mathcal{S}^M| = 57$ and $|\mathcal{S}^R| = 19$. 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 speedup indicates that the performance improvement using the designated job order. As shown in Figure 3.4(a), 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% 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 performance improvement in makespan for a better performance in total completion time. It can be noted in Figure 3.4(b) that *MK_TCT_JR* has a good *total completion time speedup*. Note that the total completion time is dominated by the positions of small jobs located in the order. 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 the Facebook workloads that have lots of small-size jobs, we will show that there is a very seriously negative impact in total completion time for *MK_JR*.

3.3.6.2 Simulation Result with Synthetic Facebook Workload

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%	1200	1
7	1600-3200	1.3%	2400	1
8	>3200	1.7%	4800	1

Table 3.3: The job size distribution at Facebook (from Table 2 in[99]) and sizes and number of jobs chosen for each bin in our synthetic workload (**SW** for short) of 50 jobs .

Table 3.3 gives a description of MapReduce jobs in production at Facebook in October 2009, provided by Zaharia et al. [99]. They are classified into 9 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 3.3 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% of the number of map tasks, in consistent with[99]. We estimate the running time of map and reduce tasks per job based on the map and reduce durations in Figure 1 of[100]. More precisely, we follow the LogNormal distribution[9] 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[92]. 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 3.3 accordingly.

Figure 3.5: Normalized simulation results for synthetic Facebook workloads under different job submission orders in a Hadoop cluster with $|\mathcal{S}^M| = 57$ and $|\mathcal{S}^R| = 19$.

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. Figure 3.5 presents the simulation results for synthetic Facebook workloads under the job ordering optimization algorithms *MK_JR* and *MK_TCT_JR*. There are about 15%~19% makespan improvement for *MK_JR* and 10%~15% for *MK_TCT_JR*. It is worth noting that the total completion time speedup in Figure 3.5(b) 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*.

3.3.6.3 Lower and Upper Bound Makespan Evaluation

Figure 3.6: The normalized comparison result of simulated makespan, lower bound makespan and upper bound makespan for optimal job order produced by *MK_JR*, as well as simulated makespan and upper bound makespan for worst-case job order based on Theorem 3.2. Particularly, the results are normalized by dividing with simulated makespan of the optimal job order.

Note that Lemma 3.2 gives a formula to compute the upper bound makespan for a MapReduce workload under an arbitrary job submission order. Moreover, Lemma 3.3 tells us how to derive a lower bound makespan as well as a upper bound makespan for all job submission orders. We perform experiments to evaluate the tightness of these lower (or upper) bound makespan with synthetic Facebook workloads. As shown in Figure 3.6, we compute the lower bound and upper bound makespan of the optimized job order produced by *MK_JR*, according to Lemma 3.2 and Lemma 3.3. In contrast, the upper bound makespan of the worst-case job order can be computed based on Theorem 3.2 and Lemma 3.2. Moreover, we derive the simulated makespan for optimal job order as well as worst-case job order with *MREstimator* to evaluate the tightness of lower and upper bound makespan. All of these results are normalized by dividing them with simulated makespan of the optimal job order. The experimental results show that, both the lower bound makespan and upper bound makespan are very close to the corresponding simulated makespan, which validates the tightness of the upper bound and lower bound makespan models.

3.3.6.4 Performance Impact of Inaccuracy in Estimating Map/Reduce Task Time

In previous sections, we take the average task execution time for map/reduce tasks of each job as input to optimize the performance of a workload. However, in practice, the execution

Figure 3.7: The performance impact of map/reduce task execution time variation for different variation percentages.

time of map/reduce tasks can be fluctuated up and down the average value, depending on specific applications. In this section, we study the impact of such a variation on the overall performance of a workload for our job ordering algorithms.

In our experiment below, we assume that it follows the uniform distribution for the variation percentage of map/reduce task execution time relative to the corresponding average task execution time. We take the testbed workload of 100 jobs and the synthetic Facebook workload of 100 jobs as examples to study the influence of different task execution time variations on makespan and total completion time.

The experimental results are shown in Figure 3.7(a) and Figure 3.7(b). The x-axis gives the maximum variation percentage (i.e., variation domain) of map/reduce task execution time relative to its average value. We generate the specific execution time for each map/reduce task with the monte carlo method that follows the uniform probability distribution for the variation percentage of execution time subject to the given variation domain. For example, Point 12 at the x-axis in Figure 3.7(a) means that the execution time for each map/reduce task fluctuates up or down randomly within 12% relative to the average task execution time with uniform probability distribution. Hence, Point 0 at the x-axis represents the performance result under the pure average task execution time. Moreover, we normalize the makespan (or total completion time) with makespan speedup (or total completion time speedup) through dividing the makespan (or total completion time) of originally unoptimized job order by the makespan of optimized job order with our job

ordering algorithms (*MK_JR* and *MK_TCT_JR*). Particularly, it is worth mentioning that the job ordering optimization performed is still based on the average task execution time. The results of Figure 3.7 show that there is a small increase for makespan (and total completion time) for both cases of unoptimized job order and optimized one with our proposed job ordering algorithms under different degrees of variation in map/reduce task execution time, whereas the relative performance improvement (i.e., makespan speedup, total completion time speedup) is fine or even better under this case. It indicates that we can take the average task execution time as input for job ordering algorithms under the case of inaccuracy in estimating map/reduce task time, assuming that the variation percentage of task execution time follows uniform probability distribution.

3.4 Performance Optimization for Online MapReduce Workloads

This section considers the performance optimization for *online* MapReduce workloads. We first present a formal definition for the problem. Then we introduce MROrder, the online job ordering optimization system for Hadoop.

3.4.1 Problem Definition

An *online* workload is a set of jobs $J = \{J_1, J_2, \dots, J_n\}$ that arrives over time. Let t_i^a denote the arriving time for J_i . 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?

3.4.2 MROrder System

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

3.4.2.1 System Overview

Figure 3.8 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

Figure 3.8: The overall architecture for *MROrder* system.

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.

3.4.2.2 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 pol-

icy 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 3.4.2.3). 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 ($n \geq n_0$). 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 is 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 3.

Algorithm 3 *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 $J_A = \{J_i | (J_i \in J) \wedge (t_i^a \leq t^{curr}) \wedge (orderflag_i = false)\}$. 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 *JobDispatch-*

ingQueue and dispatching them into the 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*, losing the impact 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. It will also have a negative impact 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 4.

Algorithm 4 *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 $orderflag_i$ for each J_i . It is initialized to be $orderflag_i = false$ 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 $J_A = \{J_i | (J_i \in J) \wedge (t_i^a \leq t^{curr}) \wedge (orderflag_i = false)\}$. 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*.
 5. The policy module updates t^{wait} as follows: $t^{wait} = \max\{\Delta t, T_A\}$, where $T_A = \max_{1 \leq k \leq |J_A|} \left\{ \sum_{i=1}^k \frac{|J_i^M| \cdot t_i^M}{|S^M|} + \sum_{i=k}^{|J_A|} \frac{|J_i^R| \cdot t_i^R}{|S^R|} \right\}$.
 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.

3.4.2.3 Job Ordering Engine

The job 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 develop 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 3.5) 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*[64], as shown in Algorithm 1 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 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 $T_i^M < T_i^R$. 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^M and those in J_B in non-increasing order of T_i^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 5, based on *shortest processing time first*. Its time complexity is $O(n \log n)$. 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 5 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:

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

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

3.4.2.4 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 adopts 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 size 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[86] 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 3.4.3.3. Generally, SIM takes longer time than ALG, but it provides better performance result. Thus, there is a tradeoff between the performance result and overhead for the dynamic choice of job ordering approach.

3.4.3 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.

3.4.3.1 Workloads

Our experiment consists of two types of synthetic workloads. One is the synthetic Facebook workload, generated based on[26,100]. The number of map/reduce tasks as and the arriving time for each job are based the input/output data sizes of workloads provided by[26]. It

contains lots of small-size jobs (more than 58% in the number of jobs)[100]. We use it primarily to evaluate the total completion time for MROrder system.

Our second workload is a testbed workload. The arriving time follows the poisson distribution[11]. 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.

3.4.3.2 Evaluation and Analysis of Policy Solutions

JobNum	TP-FTI		TP-ATI	
	$\Delta t(sec)$	PITCT (%)	$\Delta t(sec)$	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

Table 3.4: 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.

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 3.4 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 ~ 350 sec, and 10 ~ 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.

3.4.3.3 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 3.4.2.3 for details).

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 3.5: Performance and overhead comparison of ALG versus SIM.

Table 3.5 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.

(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.

Figure 3.9: The optimized performance results for MROrder system under different sizes of testbed workloads and synthetic Facebook workloads.

3.4.3.4 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 3.9 presents optimized performance results based on MROrder system, under varied sizes of online workloads. Specifically, the results for testbed workloads are shown in Figure 3.9 (a) and Figure 3.9 (b). The results for synthetic Facebook workloads are shown in Figure 3.9 (c) and Figure 3.9 (d). There is about 11% – 31% makespan improvement for testbed workloads in Figure 3.9 (a), whereas there is only 3% for Facebook workloads on average in Figure 3.9 (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 3.9 (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 dominated by the positions of small-size jobs in a workload, e.g., Facebook workload.

3.4.3.5 Accuracy Evaluation for 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

Figure 3.10: Simulated results versus experimental results for a MapReduce workload.

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 dataset² of 10GB, as application input data. We ran experiments in Amazon’s Elastic Compute Cloud(EC2)[1]. 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[1]. 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 3.10 (a) and Figure 3.10 (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*.

3.5 Summary

This chapter studies job ordering optimization for MapReduce workloads under the FIFO scheduling in a Hadoop cluster. Two performance metrics are considered, i.e., makespan and total completion time. We assume some knowledge including the number of map/reduce tasks, the execution time for each map/reduce task is known in advance. It is applicable 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. Specifically, we focus on two kinds of MapReduce workloads:

- *Offline MapReduce Workload*. We first propose a job ordering optimization algorithm for makespan. Observing that the total completion time can be poor subject to getting the optimal makespan, we further propose a new greedy job ordering optimization

²<http://dumps.wikimedia.org/enwiki/>

algorithm to minimize the makespan and total completion time together. Some theoretical results are also given for the deduction of makespan and the optimized job order.

- *Online MapReduce Workload.* We propose a prototype system named MROrder to perform job ordering optimization automatically for online MapReduce workloads. Several policy solutions are 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.

Finally, we summarize the benefits and limitations for the work in this chapter as follows:

- **Pros:** It enables the performance optimization for MapReduce workloads without modifying the Hadoop source code. It is useful for the user in a Hadoop cluster where she has no permission to modify or upgrade the Hadoop system, except for the MapReduce workloads.
- **Cons:** Due to the input needs of job's information for our algorithms and systems, they can only be used for periodically running MapReduce jobs like in data warehouse. In other words, the application generality is a problem. Second, there is an implicit assumption of a perfect cluster state for job ordering technique. That is, it does not take into account numerous practical undeterministic performance factors such as node downtown, task failure, and data skew. Third, Job ordering optimization can only improve the performance at certain degree. There is still optimization possibility/space after job ordering optimization, e.g., slot configuration optimization.

In view of this, in the next chapter, we will further optimize the performance of MapReduce workloads to overcome the severe shortcomings of job ordering approaches.

Chapter 4

Dynamic Slot Allocation Optimization for MapReduce Cluster

4.1 Introduction

In Chapter 3, we propose job ordering optimization approaches to improve the performance for MapReduce workloads without changing the Hadoop MRv1 source code. However, there are several limitations for the job ordering optimization technique as follows:

- *Lack of Generality.* Its applicability is limited to those periodical running jobs, since there are several assumptions made including knowledge of the execution time of map and reduce tasks. Moreover, it assumes that all jobs are independent and cannot apply for MapReduce workflows.
- *Insufficient Optimization.* There is still room for performance improvement beyond what has already been achieved using job ordering.
- *Limited Scheduling Support.* Job ordering optimization approach can work well for FIFO scheduling, but not for Fair Scheduler.

This chapter moves forward to optimize the performance for MapReduce workloads by resolving the aforementioned problems. In contrast to the performance optimization of job ordering from the workload aspect, we identify the limitation of Hadoop system and then modify its source code. A key challenge and interesting question to be answered in this chapter is: *how to improve the performance substantially for all kinds of MapReduce workloads (e.g., independent and dependent workloads) without any knowledge or assumption under any scheduling policy (e.g., FIFO scheduling, Fair scheduling)?*

In this chapter, we consider the performance improvement via optimizing the slot usage from two aspects: one is to maximize the map/reduce slot utilization as much as possible. Another approach is to improve the utilization efficiency of utilized slots by identifying two primary affecting factors including speculative execution and data locality optimization.

The organization of this chapter is as follows. Section 4.2 introduces our DynamicMR framework. Section 4.3 evaluates DynamicMR experimentally. We make a discussion and comparison between the job ordering optimization approach proposed in Chapter 4.1 and DynamicMR in Section 4.4. Finally, Section 4.5 summarizes the chapter.

4.2 DynamicMR

In this section, we first give an overview of the DynamicMR system. Next we introduce its three optimization techniques, namely, DHSA, SEPB, Slot PreScheduling in detail. The implementation of DynamicMR is presented in Section 4.2.5. Finally, we make a discussion on the three techniques in Section 4.2.6.

4.2.1 System Overview

In Hadoop MRv1, the compute resources (e.g., CPU cores) are abstracted into map and reduce slots, which are basic compute units and statically configured by administrator in advance. From the resource usage aspect, the slots can be categorized into two types, namely, *busy* slots (i.e., with running tasks) and *idle* slots (i.e., no running tasks). Given the total number of map and reduce slots configured by users, one optimization approach (i.e., macro-level optimization) is to improve the slot utilization by maximizing the number of busy slots and reducing the number of idle slots (Section 4.2.2). Moreover, it is worth mentioning that not every busy slot can be efficiently utilized. Thus, our further optimization method (i.e., micro-level optimization) is to improve the utilization efficiency of busy slots (Section 4.2.3 and Section 4.2.4). Particularly, we identify two major impact factors: (1). Speculative tasks (Detailed in Section 4.2.3); (2). Data locality (Detailed in Section 4.2.4). By combining the techniques of macro-level optimization and micro-level optimization, we propose DynamicMR, a dynamic slot optimization system, to improve the performance of

the Hadoop cluster. Here we introduce DynamicMR on its optimization for Hadoop Fair Scheduler. Meanwhile, it can also be applied for Hadoop FIFO scheduler.

Figure 4.1: Overview of DynamicMR System.

Figure 4.1 presents an overview of DynamicMR system. It is based on three techniques, i.e., *Dynamic Hadoop Slot Allocator (DHSA)*, *Speculative Execution Performance Balancing (SEPB)*, and *Slot PreScheduling*.

Each technique improves performance from a different aspect. Specifically, *DHSA* maximizes slot utilization while maintains the fairness. *SEPB* identifies the slot resource inefficiency problem for a Hadoop cluster, caused by speculative tasks. It works on top of the Hadoop speculative scheduler to balance the performance tradeoff between a single job and a batch of jobs. *Slot PreScheduling* improves the slot utilization efficiency and performance by improving the data locality for map tasks while keeping the fairness. It pre-schedules tasks when there are pending map tasks with data on that node, but *no allowable idle map slots* (See definitions in Section 4.2.4).

By incorporating the three techniques, it enables DynamicMR to improve the utilization and performance of a Hadoop cluster substantially with the following step-by-step optimization processes (the round labels ① ~ ③ in Figure 4.1 are corresponding to the following numbers):

- ①. *Idle Slot Allocation*. When there are idle slots, DynamicMR will first attempt to improve the slot utilization with *DHSA*. It decides dynamically whether to allocate it or not, subject to the numerous constrains, e.g., fairness, load balance.

- ②. If the allocation is true, DynamicMR further optimizes the performance by improving the efficiency of slot utilization with *SEPB*. Since the speculative execution can improve the performance of a single job but at the cost of cluster efficiency, *SEPB* acts as an efficiency balancer between a single job and a batch of jobs. It works on top of Hadoop speculative scheduler to determine dynamically whether allocating the idle slot to the pending task or speculative task.
- ③. When it comes to slot allocation for map tasks, besides delay scheduling, DynamicMR will be able to further improve the slot utilization efficiency by maximizing the data locality with *Slot PreScheduling*.

Moreover, it is worth mentioning that the three techniques are at different levels, i.e., they can work together or individually. The detailed description for each technique is presented as follows.

4.2.2 Dynamic Hadoop Slot Allocation (DHSA)

Current design of MapReduce suffers from under-utilization of the respective slots as the number of map and reduce tasks varies over time, resulting in occasions where the number of slots allocated for map/reduce is smaller than the number of map/reduce tasks. Our dynamic slot allocation policy is based on the observation that there may be idle map (or reduce) slots, as the job proceeds from map phase to reduce phase at different time periods. We can use the unused map slots for those overloaded reduce tasks to improve the performance of the MapReduce workload, and vice versa. For example, at the beginning of MapReduce workload computation, there are only computing map tasks and no computing reduce tasks, i.e., all the computation workload lies in the map-side. In that case, we can make use of idle reduce slots for running map tasks. That is, we break the implicit assumption for current MapReduce framework that the map tasks can only run on map slots and reduce tasks can only run on reduce slots, to allow that:

- *Slots are generic and can be used by either map or reduce tasks, although there is a pre-configuration for the number of map and reduce slots.* That is, when there are

insufficient map slots, the map tasks will use up all the map slots and then borrow unused reduce slots. Similarly, reduce tasks can use unallocated map slots if the number of reduce tasks is greater than the number of reduce slots.

- *Map tasks prefer to use map slots and likewise reduce tasks prefer to use reduce slots.*
- The benefit is that, the pre-configuration of map and reduce slots per slave node can still work to control the ratio of running map and reduce tasks during runtime, better than YARN that has no control mechanism for the ratio of running map and reduce tasks. The reason is that, if there is no control on the ratio of running map and reduce tasks, there is the possibility of too many reduce tasks running for data shuffling, causing the network to be a bottleneck seriously (See Section 4.3.6).

To achieve aforementioned two rules, however, there are two challenges that should be considered:

- (C1). *Fairness* is an important metric in Hadoop Fair Scheduler (HFS). We say it is *fair* when all pools have been allocated with the same amount of resources. In HFS, task slots are first allocated across the pools, and then the slots are allocated to the jobs within the pool[99]. Moreover, a MapReduce job computation consists of two parts: map-phase computation and reduce-phase computation. One question is about how to define and ensure fairness under the dynamic slot allocation policy.
- (C2). The resource requirements between the map slot and reduce slot are generally different. This is because that the map task and reduce task often exhibit completely different execution patterns. Reduce task tends to consume much more resources such as memory and network bandwidth. Simply allowing reduce tasks to use map slots requires configuring each map slot to take more resources, which consequently decreases the effective number of slots on each node, causing resource under-utilized during runtime. Thus, a careful design of dynamic allocation policy is important and needed to be aware of such difference.

With respect to (C1), we propose a *Dynamic Hadoop Slot Allocator (DHSA)*. It contains two alternatives, namely, pool-independent DHSA(PI-DHSA) and pool-dependent DHSA(PD-DHSA), each of which considers the fairness from different aspects.

4.2.2.1 Pool-Independent DHSA (PI-DHSA)

Figure 4.2: Example of the fairness-based slot allocation flow for PI-DHSA. The black arrow line and dash line show movement of slots between the map-phase pools and the reduce-phase pools.

HFS adopts max-min fairness[10] to allocate slots across pools with minimum guarantees at the map-phase and reduce-phase, respectively. *Pool-Independent DHSA (PI-DHSA)* extends the HFS by allocating slots from the cluster level, independent of pools. It considers *fair* when the numbers of typed slots allocated across typed-pools within each phase (i.e., map-phase, reduce-phase) are the same.

As shown in Figure 4.2, it presents the slot allocation flow for PI-DHSA. It is a typed phase-based dynamic slot allocation policy. The allocation procedure consists of two parts, as shown in Figure 4.2:

(1). *Intra-phase dynamic slot allocation.* Each pool is split into two sub-pools, i.e., map-phase pool and reduce-phase pool. At each phase, each pool receives its share of slots. An overloaded pool, whose slot demand exceeds its share, can dynamically borrow unused slots from other pools of the same phase. For example, an overloaded map-phase Pool 1 can borrow map slots from map-phase Pool 2 or Pool 3 when Pool 2 or Pool 3 is under-utilized, and vice versa, based on the max-min fair policy.

(2). *Inter-phase dynamic slot allocation.* After the intra-phase dynamic slot allocation for both the map-phase and reduce-phase, we can now perform dynamic slot allocation across typed phases. That is, when there are some unused reduce slots at the reduce phase, and the number of map slots at the map phase is insufficient for map tasks, it will borrow some idle reduce slots for map tasks, to maximize the cluster utilization, and vice versa.

Overall, there are four possible scenarios. Let N_M and N_R be the total number of map tasks and reduce tasks respectively, while S_M and S_R be the total number of map and reduce slots configured by users respectively. The four scenarios are below:

Case 1: When $N_M \leq S_M$ and $N_R \leq S_R$, the map tasks are run on map slots and reduce tasks are run on reduce slots, i.e., no borrowing of map and reduce slots.

Case 2: When $N_M > S_M$ and $N_R < S_R$, we satisfy reduce tasks for reduce slots first and then use those idle reduce slots for running map tasks.

Case 3: When $N_M < S_M$ and $N_R > S_R$, we can schedule those unused map slots for running reduce tasks.

Case 4: When $N_M > S_M$ and $N_R > S_R$, the system should be in completely busy state, and similar to (1), there will be no movement of map and reduce slots.

Thereby, the whole dynamic slot allocation flow is that,

Whenever a heartbeat is received from a compute node, we first compute the total demand for map slots and reduce slots for the current MapReduce workload.

Next we determine dynamically the need to borrow map (or reduce) slots for reduce (or map) tasks based on the demand for map and reduce slots, in terms of the above four scenarios. The specific number of map (or reduce) slots to be borrowed is determined based on the number of unused reduce (or map) slots and its map (or reduce) slots required.

In practice, instead of borrowing all unused map (or reduce) slots, we may often want to reserve some unused slots at each phase to minimize the possible starvation of slots for incoming MapReduce jobs. A question would be how to control the number of reserved slots dynamically.

To achieve the reservation functionality, we provide two variables *percentageOfBorrowedMapSlots* and *percentageOfBorrowedReduceSlots*, defined as the percentage of unused map and reduce slots that can be borrowed, respectively. We can thereby limit the number of unused map and reduce slots that should be allocated for map and reduce tasks at this heartbeat for that tasktracker.

Moreover, Challenge (C2) reminds us that we cannot treat map and reduce slots the same, and simply borrow unused slots for map and reduce tasks. Instead, we should be

aware of varied resource sizes of map and reduce slots. A slot weight-based approach is thus proposed to address the problem. We assign the map and reduce slots with different weight values, in terms of the resource configurations. Based on the weights, we can dynamically determine how much map and reduce tasks should be spawn during runtime. For example, consider a tasktracker with the map/reduce slot configuration of 8/4. According to varied resource requirements, let's assume that the weights for map and reduce slots are 1 and 2, respectively. Thus, the total resource weight is $8 \times 1 + 4 \times 2 = 16$. With slot weight-based approach for dynamic borrowing, the maximum number of running map tasks can be 16 in that compute node, whereas the number of running reduce tasks is at most $8/2 + 4 = 8$ rather than 16.

Finally, the details of DHSA are shown in Algorithm 6, with the time complexity of $O(1)$.

Algorithm 6 The dynamic task assignment policy for tasktracker under PI-DHSA.

When a heartbeat is received from a compute node n :

- 1: compute its clusterUsedMapSlots, clusterUsedReduceSlots, mapSlotsDemand, reduceSlotsDemand, mapSlotsLoadFactor and reduceSlotsLoadFactor.
 - 2: */*Case 1: both map slots and reduce slots are sufficient.*/*
 - 3: **if** (mapSlotsLoadFactor ≤ 1 and reduceSlotsLoadFactor ≤ 1) **then**
 - 4: *//No borrow operation is needed.*
 - 5: **end if**
 - 6: */*Case 2: both map slots and reduce slots are not enough.*/*
 - 7: **if** (mapSlotsLoadFactor ≥ 1 and reduceSlotsLoadFactor ≥ 1) **then**
 - 8: *//No borrow operation is needed.*
 - 9: **end if**
 - 10: */*Case 3: map slots are enough, while reduce slots are insufficient. It calculates borrowed map slots for reduce tasks.*/*
 - 11: **if** (mapSlotsLoadFactor < 1 and reduceSlotsLoadFactor > 1) **then**
 - 12: currentBorrowedMapSlots = clusterUsedMapSlots - clusterRunningMapTasks;
 - 13: extraReduceSlotsDemand = min{ max{ floor{ clusterMapCapacity * percentageOfBorrowedMapSlots } - currentBorrowedMapSlots, 0 }, reduceSlotsDemand - clusterReduceCapacity }
 - 14: updatedMapSlotsLoadFactor = (mapSlotsDemand + extraReduceSlotsDemand) / clusterMapCapacity;
 - 15: **end if**
 - 16: */*Case 4: map slots are insufficient, while reduce slots are enough. It calculates borrowed reduce slots for map tasks.*/*
 - 17: **if** (mapSlotsLoadFactor > 1 and reduceSlotsLoadFactor < 1) **then**
 - 18: currentBorrowedReduceSlots = clusterUsedReduceSlots - clusterRunningReduceTasks;
 - 19: extraMapSlotsDemand = min{ max{ floor{ clusterReduceCapacity * percentageOfBorrowedReduceSlots } - currentBorrowedReduceSlots, 0 }, mapSlotsDemand - clusterMapCapacity }
 - 20: updatedReduceSlotsLoadFactor = (reduceSlotsDemand + extraMapSlotsDemand) / clusterReduceCapacity;
 - 21: **end if**
 - 22: compute availableMapSlots and availableReduceSlots based on the updated map/reduce load factor and used slots.
-

4.2.2.2 Pool-Dependent DHSA (PD-DHSA)

Figure 4.3: Example of the fairness-based slot allocation flow for PD-DHSA. The black arrow line and dash line show the borrow flow for slots across pools.

In contrast to PI-DHSA that considers the the fairness in its dynamic slot allocation independent of pools, Pool-Dependent DHSA (PD-DHSA) considers another fairness for the dynamic slot allocation across pools, as shown in Figure 4.3. It assumes that each pool, consisting of two parts: map-phase pool and reduce-phase pool, is selfish. That is, it always tries to satisfy its own shared map and reduce slots for its own needs at the map-phase and reduce-phase as much as possible before lending them to other pools. It considers *fair* when total numbers of map and reduce slots allocated across pools are the same with each other. PD-DHSA will be done with the following two processes:

(1). *Intra-pool dynamic slot allocation*. First, each typed-phase pool will receive its share of typed-slots based on max-min fairness at each phase. There are four possible relationships for each pool regarding its demand (denoted as *mapSlotsDemand*, *reduceSlotsDemand*) and its share (marked as *mapShare*, *reduceShare*) between two phases:

Case (a). $mapSlotsDemand < mapShare$, and $reduceSlotsDemand > reduceShare$. We can borrow some unused map slots for its overloaded reduce tasks from its reduce-phase pool first before yielding to other pools.

Case (b). $mapSlotsDemand > mapShare$, and $reduceSlotsDemand < reduceShare$. In contrast, we can satisfy some unused reduce slots for its map tasks from its map-phase pool first before giving to other pools.

Case (c). $mapSlotsDemand \leq mapShare$, and $reduceSlotsDemand \leq reduceShare$. Both map slots and reduce slots are enough for its own use. It can lend some unused map slots and reduce slots to other pools.

Case (d). $mapSlotsDemand > mapShare$, and $reduceSlotsDemand > reduceShare$. Both map slots and reduce slots for a pool are insufficient. It might need to borrow some unused map or reduce slots from other pools through *inter-Pool dynamic slot allocation* below.

(2). *Inter-pool dynamic slot allocation*. It is obvious that, (i). for a pool, when its $mapSlotsDemand + reduceSlotsDemand \leq mapShare + reduceShare$. The slots are enough for the pool and there is no need to borrow some map or reduce slots from other pools. It is possible for the cases: (a), (b), (c) mentioned above. (ii). On the contrary, when $mapSlotsDemand + reduceSlotsDemand > mapShare + reduceShare$, the slots are not enough even after *Intra-pool dynamic slot allocation*. It will need to borrow some unused map and reduce slots from other pools, i.e., *Inter-pool dynamic slot allocation*, to maximize its own need if possible. It can occurs for pools in the following cases: (a), (b), (d) above.

The overall slot allocation process for PD-DHSA is as follows:

When a tasktracker receives a heartbeat, instead of allocating map and reduce slots separately, it treats them as a whole during the allocation for pools. It first computes the maximum number of free slots that can be allocated at each round of heartbeat for the tasktracker. Next it starts the slot allocation for pools. For each pool, there are four possible slot allocations as illustrated in Figure 4.4 below (The number labeled in the graph denotes the corresponding case):

Case (1): We first try the map tasks allocation if there are idle map slots for the tasktracker, and there are pending map tasks for the pool.

Case (2): If the attempt of Case (1) fails since the condition does not hold or it cannot find a map task satisfying the valid data-locality level, we continue to try reduce tasks allocation when there are pending reduce tasks and idle reduce slots.

Case (3): If Case (2) fails due to the required condition does not hold, we try for map task allocation again. Case (1) fails might be that there are no idle map slots available. In

contrast, Case (2) fails might be due to no pending reduce tasks. In this case, we can try reduce slots for map tasks of the pool.

Case (4): If Case (3) fails, we try for reduce task allocation again. Case (1) and Case (3) fail might be due to no valid locality-level pending map tasks available, whereas there are idle map slots. In contrast, Case (2) might be that there are no idle reduce slots available. In that case, we can allocate map slots for reduce tasks of the pool.

Moreover, there is a special scenario that needs to be considered. Note it is possible that all the above four possible slot allocation attempts fail for all pools, due to the data locality consideration for map tasks. For example, it is possible that there is a new compute node added to the Hadoop cluster. It may be empty and does not contain any data. Thus, the data locality for all map tasks might not be satisfied and all pending map tasks cannot be issued. The failures of both Case (2) and Case (4) indicate that there are no pending reduce tasks available for all pools. However, there might be some pending map tasks. Therefore, there is a need to run some map tasks by ignoring the data locality consideration on that new compute node to maximize the system utilization. To implement this, we make a mark *visitedForMap* for each job visited for map tasks. The data locality will be considered when *visitedForMap* does not contain scanned job. Otherwise, it will relax the data locality constrain for map tasks.

Finally, The detailed implementation for PD-DHSA is given in Algorithm 7, whose time complexity is $O(n^2)$.

4.2.2.3 Discussion on DHSA

DHSA VS Static Slot Optimization. Traditional Hadoop MRv1 adopts the static slot configuration method for map and reduce tasks and has a strict constrain that map tasks can only use map slots and reduce tasks can only run on reduce slots, making the performance sensitive to the slot configurations, i.e., different slot configurations will results in varied performance, as illustrated in Figure 4.10. Moreover, for FIFO scheduling, different job submission orders also have varied performance[89]. In contrast, for Fair scheduling, different weights (i.e., shares) of pools can cause varied performance for the whole jobs.

Figure 4.4: The slot allocation flow for each pool under PD-DHSA. The numbers labeled in the graph corresponds to Case (1)-(4) in Section 4.2.2.2, respectively.

It implies that, we can improve the performance for Hadoop MRv1 statically via optimizing slot configuration, job submission orders, and pool weights. Compared to DHSA, such static methods have the following serious shortcomings:

First, the static optimization methods need to get some information about MapReduce workloads (e.g., the number of map/reduce tasks per job/pool, the execution time for map/reduce tasks per job) in advance, which is however often un-obtainable without running tasks in practice. Instead, it can only be used for periodically running jobs (e.g., in data warehouse). Thus, there is a generality problem for static methods. In contrast, DHSA improves the performance without needs of any information about MapReduce jobs in advance and thus keeps the generality feature of Hadoop.

Second, the static optimization methods can improve the performance of MapReduce jobs. However, static slot configuration will never be optimal as the number of Map and reduce tasks varies over time. Consider the static slot configuration for example, no matter how to optimize the slot configuration per slave node, it still cannot avoid the case that there are pending tasks and idle slots (e.g., for a single job case, the map slots will be idle in the reduce-phase computation). In contrast, with DHSA, we can guarantee that all slots must be busy whenever there are pending tasks, making slot utilization at maximum.

Algorithm 7 The dynamic task assignment policy for tasktracker under PD-DHSA.**When** a heartbeat is received from tasktracker *tt*s:

```

1: Compute its totalSlotsDemand, totalSlotsCapacity, trackerSlotsCapacity, trackerRunningTasksNum and
   trackerCurrentSlotsCapacity.
2: /* Return when there are no idle slots. */
3: if trackerRunningTasksNum  $\geq$  trackerCurrentSlotsCapacity then
4:   return NULL;
5: end if
6: for (i = 0; i < trackerCurrentSlotsCapacity - trackerRunningTasksNum; i++) do
7:   Sort pools by distance below min and fair share
8:   for (Pool p : pools) do
9:     /* Case (1): allocate map slots for map tasks from Pool p */
10:    if (there are pending map tasks and idle map slots) then
11:      attempt to allocate map slots for map tasks (considering data locality) and jump out of loop if
      allocation succeeded.
12:    end if
13:    /* Case (2): allocate reduce slots for reduce tasks from Pool p */
14:    if (Case (1) failed and there are pending reduce tasks and idle reduce slots) then
15:      attempt to allocate reduce slots for reduce tasks and jump out of loop if allocation succeeded.
16:    end if
17:    /* Case (3): allocate reduce slots for map tasks from Pool p */
18:    if (Case (2) failed and there are pending map tasks) then
19:      attempt to allocate reduce slots for map tasks (considering data locality) and jump out of loop
      if allocation succeeded.
20:    end if
21:    /* Case (4): allocate map slots for reduce tasks from Pool p */
22:    if (Case (3) failed and there are pending reduce tasks) then
23:      attempt to allocate map slots for reduce tasks and jump out of loop if allocation succeeded.
24:    end if
25:  end for
26:  /* Case (5): schedule the non-local map tasks when its node-local tasks cannot be satisfied. */
27:  if (Case (1)-(4) failed) then
28:    for (Pool p : pools) do
29:      if (there are pending map tasks) then
30:        attempt to allocate map/reduce slots to map tasks (not considering data locality) and jump
        out of loop if allocation succeeded.
31:      end if
32:    end for
33:  end if
34: end for

```

Dynamic slot allocation. Underlying PI-DHSA and PD-DHSA are two different concepts or definitions of fairness. PI-DHSA follows strictly the definition of fairness given by traditional HFS, i.e., the slots are fairly shared across pools within each phase (e.g., map-phase, reduce-phase), but independent across phases. In contrast, PD-DHSA gives a new definition of fairness from the perspective of pools, i.e., each pool shares the total number of map and reduce slots from the map phase and reduce phase fairly with other pools. Due to varied definitions of fairness, there are different priorities and possibilities for slot movements between map-phase and reduce-phase (i.e., moving map slots to reduce-phase

for reduce tasks, and vice versa) for PI-DHSA and PD-DHSA. For PI-DHSA, the map slots always satisfy the map tasks first before giving to reduce tasks, and vice versa. Thus, the inter-phase slot movement can only occur when one type of slots (e.g., map slots) are enough while the other type of slots (e.g., reduce slots) are insufficient, i.e., two cases: Case 2 and Case 3 in Section 4.2.2.1. In contrast, for PD-DHSA, all the slots of a pool always attempt to satisfy the tasks within the pool first before yielding to other pools. The inter-phase slot movement appears when it exists a pool whose one type of slots (e.g., map slots) share is larger than its corresponding type of slots demand while its share of the other type of slots (e.g., reduce slots) is not enough. However, all four scenarios (e.g., Case 1, Case 2, Case 3, Case 4) mentioned in Section 4.2.2.1 could have such a kind of pools, implying that the inter-phase slot movement can occur in all the four scenarios. That is, it is more likely to have inter-phase slot movement for PD-DHSA than PI-DHSA during the dynamic slot allocation process.

4.2.3 Speculative Execution Performance Balancing (SEPB)

MapReduce job's execution time is very sensitive to slow-running tasks (namely *straggler*)[97]. There are various reasons that cause stragglers, including faulty hardware and software mis-configuration[102]. We classify the stragglers into two types, namely, *Hard Straggler* and *Software Straggler*, defined as follows:

- **Hard Straggler:** A task that goes into a deadlock status due to the endless waiting for certain resources. It will continue running until the tasks is terminated.
- **Soft Straggler:** A task that can complete its computation successfully, but will take much longer time than the common tasks.

For the hard straggler, we should kill it and run another equivalent task, or called a *backup task*, immediately once it was detected. In contrast, there are two possibilities between the soft straggler and its backup task:

(S1). Soft straggler completes first or the same as its backup task. For this case, there is no need to run a backup task.

(S2). Soft straggler finishes later than the backup task. We should terminate it and run a backup task immediately.

To deal with the straggler problem, *speculative execution* is used in Hadoop. Instead of diagnosing and fixing straggling tasks, it detects the straggling task dynamically using heuristic algorithms such as LATE[102]. Once detected, however, it cannot simply kill the straggler immediately due to the following facts:

- Hadoop does not have a mechanism to distinguish between the hard straggler and the soft straggler;
- Moreover, for the soft straggler, it's also difficult to judge whether it belongs to (S1) or (S2) before running. Simply killing the straggler will harm the case of (S1).

Instead, it spawns a backup task and allows it to concurrently run with the straggler, i.e., there is a computation overlap between the straggler and the backup task. The task killing operation will occur when either of the two tasks completed. It is worth mentioning that, although the speculative execution can reduce a single job's execution time, but it comes at the cost of cluster efficiency[97]. Speculative tasks are not free, i.e., they compete for certain resources, including map slots, reduce slots and network, with other running tasks of jobs, which could have a negative impact for the performance of a batch of jobs. Therefore, it arises a challenge issue for speculative tasks on how to mitigate its negative impact for the performance of a batch of jobs.

To maximize the performance for a batch of jobs, an intuitive solution is that, given an idle task slot, we should satisfy pending tasks first before considering speculative tasks. That is, when a node has an idle map slot, we should choose pending map tasks first before looking for speculative map tasks for a batch of jobs. Moreover, recall that in our dynamic scheduler, the map slot is no longer restricted to map task, it can serve reduce task. It means that, for an empty map slot, we should consider choosing pending map task first, then pending reduce task, later speculative map task, and finally speculative reduce task if we want to further maximize the performance for a batch of jobs, and vice versa.

We propose a dynamic task allocation mechanism called *Speculative Execution Performance Balancing (SEPB)* for a batch of jobs with speculative execution tasks on top of Hadoop's current task selection policy. Hadoop chooses a task from a job based on the following priority: first, any failed task is given the highest priority. Second, the pending tasks are considered. For map, tasks with data local to the compute node are chosen first. Finally, Hadoop looks for a straggling task to execute speculatively. In our task scheduling mechanism, we define a variable *percentageOfJobsCheckedForPendingTasks* with value domain between 0.0 and 1.0, configurable by users, to control *maxNumOfJobsCheckedForPendingTasks*, which is the maximum number of jobs that are checked for pending map and reduce tasks for a batch of jobs, as shown in Figure 4.5. Users can balance the trade-off between the performance for a batch of jobs and a single job's response time, with speculative task execution. Better performance for the whole job is obtained when *percentageOfJobsCheckedForPendingTasks* is large, Otherwise it will be better for a single job's response time. The detail of our mechanism is that, when there is an idle map slot, we first check jobs $\{J_1, J_2, J_3, \dots\}$ for map tasks. For each job J_i , we compute the total number of pending map and reduce tasks by scanning all jobs between J_i and J_j , where $i = 1, 2, 3, \dots, j = i + \text{maxNumOfJobsCheckedForPendingTasks} - 1$. Next, we check each job J_i for the following conditions:

- (1). No failed map tasks and pending map tasks for job J_i ;
- (2). The total number of pending map tasks is larger than zero;
- (3). The total number of pending reduce tasks is larger than zero, and *percentageOfBorrowedMapSlots* is larger than zero.

The job J_i will be skipped for looking for speculative map tasks when either Conditions (1) and (2), or Condition (1) and (3) is satisfied. Otherwise, we will scan it for possible speculative map tasks, with Hadoop's speculative task mechanism [97], or *LATE*[102].

However, delaying the scheduling of speculative task will bring another challenging problem. For the hard straggler or the soft straggler of (S2) that occupies a slot for a really long time, if we do not solve it as early as possible, then the resource allocated to the slot are being used inefficiently, hence reducing the efficiency of the cluster.

To address this problem, we use a simple heuristic algorithm: We estimate the execution time for each task. When it took twice longer than the average execution time of tasks, we kill it directly to yield the slot. Since failed/killed tasks have the highest priority to run in Hadoop, a backup task will be created to replace it quickly, improving the performance of a single job and mitigating the negative impact on the cluster efficiency.

Finally, speculative reduce tasks are handled similarly. The detailed implementation of SEPB is given in Algorithm 8, with time complexity of $O(n)$.

Figure 4.5: The computation policy for the *totalNumOfPendingMapTasks* and *totalNumOfPendingReduceTasks*. It determines whether it is time to schedule speculative map tasks for current job or not.

Algorithm 8 The pseudo-code for SEPB.

When there is a idle slot:

- 1: The speculative scheduler (e.g., LATE) checks first whether there are new straggled tasks.
 - 2: **if** there are straggled tasks and *runningSpeculativeTasks* < *SpeculativeCap* **then**
 - 3: Check a set of multiple jobs for pending tasks.
 - 4: **if** there are pending tasks for multiple jobs **then**
 - 5: Allocate the idle slot to a pending task.
 - 6: **else**
 - 7: Create a speculative task and allocate the idle slot to it.
 - 8: **end if**
 - 9: **else**
 - 10: Allocate the idle slot to a pending task.
 - 11: **end if**
-

4.2.4 Slot PreScheduling

Keeping the task computation at the computing node with the local data (i.e., Data locality) is an efficient and important approach to improve the efficiency of slot utilization and performance. *Delay Scheduler* has been proposed to improve the data locality in MapReduce by [100]. It delays the scheduling for a job by a small amount of time, when it detects there

are *no local* map tasks from that job on a node where its input data reside. However, it is at the expense of fairness. There is a tradeoff between the data locality and fairness optimization with delay scheduler. It means that, in HFS, delay scheduling is not enough and there is still room for data locality improvement. A challenging question will be: Are there any solutions that can further improve the data locality while have no impact on fairness?

To answer this question, we propose a *Slot PreScheduling* technique that can improve the data locality while having no negative impact on the fairness of MapReduce jobs. In contrast to delay scheduler, it is achieved at the expense of load balance across slave nodes. The basic idea is that, in light of the fact that there are often some idle slots which cannot be allocated due to the load balancing constrain during runtime, we can pre-allocate those slots of the node to jobs to maximize the data locality.

4.2.4.1 Preliminary

In Hadoop task scheduling, there is a load manager that attempts to balance the workload across slave nodes, making the ratios of used resources close to each other among slave nodes (i.e., load balancing). Prior to presenting Slot PreScheduling, we start with two definitions:

Definition 1 The **allowable idle map (or reduce) slots** refer to the maximum number of idle map (or reduce) slots that are allowed to be allocated for a tasktracker, considering the load balancing between machines.

Definition 2 The **extra idle map (or reduce) slots** refer to the remaining idle map (or reduce) slots by subtracting the maximum value of used map (or reduce) slots and *allowable idle map (or reduce) slots* from the total number of map slots for a tasktracker, considering the load balancing between machines.

As illustrated in Figure 4.6, the workload balancing line shows the current number of map slots that can be used under an ideal balanced case. The *allowable idle map slots* are illustrated by the white area below the workload balancing line. We can note that Tasktracker 1 and 3 have some allowable idle map slots, whereas there are no *allowable idle map slots* available for Tasktracker 2 and 4. In contrast, all tasktrackers have *extra idle map slots* (See **Definition 2**), which are illustrated with the white area above the workload balancing line.

Figure 4.6: An example of unbalanced workload distribution of running map tasks in a Hadoop cluster. We assume the current priority order of fair-share allocation is $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$.

4.2.4.2 Observation and Optimization

In practice, for a MapReduce cluster, the computing workloads of running map (or reduce) tasks between tasktrackers (i.e., machines) are generally varied, because of the following facts.

- (1). Lots of MapReduce clusters in real world consist of *heterogeneous* machines (i.e., different computing powers between machines),
- (2). There are often varied computing loads (i.e., execution time) for map and reduce tasks from different jobs, due to the varied input data sizes as well as applications,
- (3). Even for a single MapReduce job under the homogenous environment, the execution time for map (or reduce) tasks may still not be the same, due to the *skew* caused by an uneven distribution of input data to tasks and some portions of the input data might take longer time to process than others[69],

For example, Figure 4.6 illustrates an *unbalanced* workload distribution of running map tasks in a Hadoop cluster, consisting of two racks each with two machines. To balance the workload, Hadoop provides a mechanism to dynamically control the number of *allowable idle map (or reduce) slots* (See **Definition 1**) for a tasktracker in a heartbeat as the following three steps.

Step 1#: Compute the load factor *mapSlotsLoadFactor* as the sum of pending map tasks and running map tasks from all jobs divided by the cluster map slot capacity.

Step 2#: Compute the current maximum number of usable map slots by multiplying $\min\{mapSlotsLoadFactor, 1\}$ with the number of map slots in a tasktracker.

Step 3#: Finally, we can compute the current *allowable idle map (or reduce) slots* for a tasktracker, by subtracting the current number of used map (or reduce) slots from the current maximum number of usable map slots.

Let's suppose that there are four running jobs and two replicas for each block data in Figure 4.6. Let's assume the priority order of fairness allocation is $J_1 \rightarrow J_2 \rightarrow J_3 \rightarrow J_4$. Under the current load distribution, we can see that there are no *allowable idle map slots* for all those tasktrackers (e.g., Tasktracker 2 and 4) with local block data of J_1 . It means that, based on the delay scheduling, J_1 will be delayed to schedule within a time limit, at the expense of fairness, no matter which tasktracker connects to the jobTracker in a heartbeat. However, we can see that there are idle map slots for each tasktracker. If we relax the strict load balancing constrain when Tasktracker 2 or 4 connects to the Jobtracker, we can proactively allocate the *extra idle map slots* to J_1 , satisfying both the data locality and fairness requirements. Based on this observation, we propose a scheduler named ***Slot PreScheduling*** to proactively allocate slots to those jobs with local map tasks, aiming to achieve both the data locality maximization and fairness requirement.

There are two cases for using Slot PreScheduling, as shown in Figure 4.7. The first case considers a tasktracker *tt*s on which there are extra idle map slots available, but no allowable idle map slots. For a headed job following the fair-share priority order, when it has local map tasks with block data on the *tt*s, instead of skipping it by the default Hadoop scheduler, we can proactively allocate extra map slots to the job.

The second case is for DHSA. When there are no idle map slots but some idle reduce slots available, for a connected tasktracker *tt*s in a heartbeat, we can proactively borrow idle reduce slots for local pending map tasks and restore them later, in order to maximize the data locality.

Figure 4.7: The scenarios for using Slot PreScheduling.

4.2.4.3 Design and Implementation of Slot PreScheduling

Figure 4.8 gives an overview design and implementation of Slot PreScheduling and its incorporation with PD-DHSA. It is an extended graph on top of Figure 4.4. Given a task-tracker *tts* connecting to the jobTracker in a heartbeat, the overall execution flow is as follows:

Case (P1): The slot allocation work starts from Case (1) of PD-DHSA. When the attempt of Case (1) fails, instead of going forwards to Case (2) as in Figure 4.4, it comes to Case (P1) of Slot PreScheduling. It checks whether the condition of *Case 1* in Figure 4.7 holds. If yes, the Slot PreScheduling will perform the map task assignment for the current job. Otherwise, it checks whether the condition of *Case 2* in Figure 4.7 holds or not, by moving towards to Case (P2).

Case (P2): It checks whether there are idle reduce slots and pending map tasks for the current job with data on *tts*. If the condition holds, it attempts to perform map task assignment work on *tts* by borrowing idle reduce slots. Moreover, there is a variable *borrowedReduceSlots*, which counts the current number of reduce slots that has been borrowed by Slot PreScheduling. To limit the maximum number of idle reduce slots that can be borrowed by Slot PreScheduling, we provide user with a configurable threshold variable *maximumBorrowableReduceSlots*, initialized to be infinity by default. The Slot PreScheduling can proactively perform map task assignment if and only if *borrowedReduceSlots* does not

Figure 4.8: The overview design and implementation of Slot PreScheduling and its combination with PD-DHSA. The labels (1)-(4), (P1)-(P3) in the graph corresponds to Case (1)-(4) in Section 4.2.2.2 and Case (P1)-(P3) in Section 4.2.4.3, respectively.

exceed *maximumBorrowableReduceSlots*. Otherwise, the Slot PreScheduling will deliver the task assignment work to PD-DHSA (i.e., Case (2)).

Case (P3): When the condition for Case (2) of PD-DHSA does not hold, Slot PreScheduling will begin to work by checking whether it is possible to assign pending reduce tasks with idle map slots in case that the variable *borrowedReduceSlots* is larger than zero. If yes, Slot PreScheduling will perform reduce task assignment with idle map slots.

Finally, Algorithm 9 shows the detailed implementation of Slot PreScheduling technique. Its time complexity is $O(n)$

Algorithm 9 The pseudo-code for Slot PreScheduling.

When a heartbeat is received from a compute node n :

```

1: Compute allowableIdleMapSlots, extraIdleMapSlots, idleReduceSlots.
2: for (Job  $j$  in job set  $J$  of sorted fair-share priority order) do
3:   if Job  $j$  has local data in node  $n$  but no allowableIdleMapSlots then
4:     if extraIdleMapSlots > 0 then
5:       Allocate map slot to Job  $j$ .
6:     end if
7:     if EnableDHSA=true && idleReduceSlots > 0 then
8:       Allocate map slot to Job  $j$ .
9:     end if
10:  end if
11: end for

```

▷ The condition to consider Slot PreScheduling.
 ▷ Case 1: using *extraIdleMapSlots*
 ▷ Case 2: using *idleReduceSlots* in DHSA.

4.2.4.4 Comparison with Delay Scheduler

In this section, we make a comparison and discussion between Delay Scheduler and Slot PreScheduling. First, they consider completely opposite scenarios. That is, Slot PreScheduling works for the case when there are pending map tasks for the current job with local block data on the tasktracker *tts*, whereas Delay Scheduler considers the case without local pending map tasks. Based on this fact, in our work, we combine them together to maximise data locality. For example, in Figure 4.6, when the current connected tasktracker in a heartbeat is Tasktracker 1 or 3, the *Delay Scheduler* will work to delay the scheduling of J_1 , to improve the data locality. In contrast, when either Tasktracker 2 or 4 connects to the jobTracker, the *Slot PreScheduling* will work by allocating the extra idle map slots to J_1 , improving the data locality and guaranteeing fairness.

Table 4.1 lists the benefits and costs for Slot PreScheduling and Delay Scheduler under different metrics, including Fairness, Data Locality and Load Balance. We can see that, the Slot PreScheduling can benefits (or improves) both fairness and data locality metrics, but at the expense of load balance, since it uses the extra idle map slots. However, for Delay Scheduler, it is favorable for data locality and load balance, whereas at the cost of fairness.

	Fairness	Data Locality	Load Balance
Slot PreScheduling	+	+	–
Delay Scheduler	–	+	+

Table 4.1: Benefit and Cost Comparison between Slot PreScheduling and Delay Scheduler. ‘+’ denotes the benefit, while ‘–’ represents the cost (or expense).

4.2.5 DynamicMR Implementation

We implement DynamicMR based on HFS source code. Table 4.2 lists some key functions for DynamicMR. We have a taskScheduler class *DynamicMRFairScheduler*, acting as the core role for task scheduling. For the DHSA component, it provides two functions *assignTasks_poolInDependentMode()* and *assignTasks_poolDependentMode()* implemented for PI-DHSA and PD-DHSA, respectively. Moreover, there are two functions *prescheduler_mapTask()* and *prescheduler_reduceTask()* for map tasks and reduce tasks

Table 4.2: The key functions in the DynamicMR.

Class	Function Description
DynamicMRFairScheduler	<code>List<Task> assignTasks_poolIndependentMode()</code> Assign map/reduce tasks across pools dynamically with PI-DHSA. See Algorithm 6 for details.
	<code>List<Task> assignTasks_poolDependentMode()</code> Assign map/reduce tasks across pools dynamically with PD-DHSA. See Algorithm 7 for details.
	<code>ArrayList<Task> prescheduler_mapTask()</code> Preschedule <i>local</i> map tasks using idle map and reduce slots, for Case <i>P1</i> and <i>P2</i> at <i>Design and Implementation</i> in Section 4.2.4.3.
	<code>ArrayList<Task> prescheduler_reduceTask()</code> Preschedule reduce tasks using map slots for Case <i>P3</i> illustrated in Figure 4.8 in Section 4.2.4.3.
PoolSchedulable	<code>Task task DynamicSpeculativeTaskScheduler()</code> Schedule speculative tasks dynamically for utilization efficiency. See detailed implementation in Section 4.2.4.3.

Table 4.3: The user configurable arguments for DynamicMR.

Types	Argument	Location
DHSA	<code>mapred.idle.mapslots.borrowedforReduceTasks</code> Fraction of the number of unused map slots (0.0 ~ 1.0) that can be used for reduce tasks, when map slots is not enough.	mapred-site.xml
	<code>mapred.idle.reduceslots.borrowedforMapTasks</code> Fraction of the number of unused reduce slots (0.0 ~ 1.0) that can be used for map tasks, when reduce slots is not enough.	
	<code>PoolsFairShareSlotsAdaptiveMode</code> The choice of DHSA. For PI-DHSA, set it to <i>PoolIndependent</i> ; and otherwise configure it with <i>PoolDependent</i> for PD-DHSA.	fair-scheduler.xml
SEPB	<code>mapred.submitted.jobs.checkedforPendingTasks</code> Fraction of the total number of jobs (0.0 ~ 1.0) to be checked for the number of pending map (reduce) tasks at cluster-level, needed by DynamicMR to decide whether it is time to run the speculative task or not for a batch of jobs.	mapred-site.xml
	<code>poolPercentageOfSubmittedJobsCheckedForPendingTasks</code> Fraction of the total number of jobs within a pool (0.0 ~ 1.0) to be checked for the number of pending map (or reduce) tasks, needed by DynamicMR to decide whether it is time or not to run the speculative task for a batch of jobs from a pool.	fair-scheduler.xml
	<code>FairSchedulerSpeculativeJobsPendingTasksCheckedMode</code> Set it to be <i>GlobalLevel</i> when we check pending tasks for all jobs in a cluster. Otherwise, set it to be <i>PoolLevel</i> if we want to check pending tasks for all jobs within each pool.	
Slot	<code>mapred.map.tasks.preScheduler.enabled</code> If <i>true</i> , it preSchedules local map tasks on the node when there are no allowable map slots.	mapred-site.xml
PreScheduling	<code>mapred.preScheduler.maximumNumBorrowedReduceSlots</code> The maximum number of reduce slots that can be borrowed for map tasks with Slot PreScheduling, 100000 by default.	

preScheduling, respectively. Figure 4.7 gives an illustration on when to use them. In contrast, for SEPB, we implement it in PoolSchedulable class. *DynamicSpeculativeTaskScheduler()* dynamically determines when to schedule speculative tasks for utilization efficiency optimization.

As mentioned at the beginning of Section 4.2 before, the three techniques, DHSA, SEPB, Slot PreScheduling are loosely coupled. They can work alone or together. To support users' needs on the three techniques, we provide users with some configurable arguments as shown in Table 4.3. For DHSA, users can choose the types (e.g., PI-DHSA, PD-DHSA) they want to use. Meanwhile, users can specify the maximum number of idle map (reduce) slots (e.g., *mapred.idle.mapslots*.

borrowedforReduceTasks, *mapred.idle.reduceslots.borrowedforMapTasks*) that can be borrowed for reduce (map) tasks at a moment. The DHSA can be disabled by simply setting them to zero. For SEPB, we provide users with two alternative modes (i.e., cluster-level and pool-level), specified by *FairSchedulerSpeculativeJobsPendingTasksCheckedMode*. Instead of checking all pending tasks for all jobs, we allow users to specify how many percentage of runnable jobs they want to check for pending tasks before allowing to schedule speculative tasks (e.g., *poolPercentageOfSubmittedJobsCheckedForPendingTasks* and *poolPercentageOfSubmittedJobsCheckedForPendingTasks*). It can be disabled by configuring them with zero. Finally, for Slot PreScheduling, users can enable it by setting *mapred.map.tasks.SlotPreScheduling.enabled* to be *true*. We also allow users to specify the maximum number of reduce slots (e.g., *mapred.preScheduler.maximumNumBorrowedReduceSlots*) that can be preempted by map tasks for data locality improvement.

4.2.6 Discussion

The goal of our work is to improve the performance for MapReduce workloads while maintaining the fairness across pools when HFS is adopted. To achieve this goal, we propose a framework called DynamicMR, consisting of three different dynamic slot allocation policies, i.e., DHSA, SEPB, Slot PreScheduling.

Table 4.4 summarizes the comparison results for the three policies with respect to different metrics (e.g., fairness, slot utilization, and performance). First, all the three policies

are favorable for the performance improvement of MapReduce workloads, due to the benefits from slot utilization optimization. Specifically, DHSA improves the performance by increasing the slot utilization. In contrast, rather than attempting to improve the slot utilization, SEPB and Slot PreScheduling achieve the performance improvement by maximizing the efficiency of slot utilization, under a given slot utilization. For fairness metric, both DHSA and DS (i.e., Delay Scheduler) have an impact on it, whereas SEPB does not. Specifically, PI-DHSA, PD-DHSA and PS (i.e., Slot PreScheduling) can benefit fair sharing, whereas DS has a negative impact on fairness.

Moreover, YARN, as the second generation of Hadoop, also aims to improve the cluster resource utilization. There are no static preconfigured slots in YARN. Each map or reduce tasks are run in containers which are spawned dynamically as required. From this point of view, both DHSA of DynamicMR and YARN are both able to maximize the resource utilization. However, DHSA is superior to YARN in performance due to its control mechanism over the ratio of map to reduce tasks, as the experimental results illustrated in Section 4.3.6. YARN currently allows no such control and could spawn more reduce tasks than the network can handle. Second, SEPB is an enhancement technique over YARN to handle the negative impact of straggler tasks on the performance. Third, Slot Prescheduling can also be Incorporated into YARN to maximize the data locality.

Techniques		Fairness	Slots Utilization	Performance
DHSA	PI-DHSA	+	+	+
	PD-DHSA			
SEPB			%(+)	+
DS		—	%(+)	+
SPS		+		

Table 4.4: Benefit and cost comparison for allocation components regarding each metric. 'DS' is an abbreviation for Delay Scheduler, while 'SPS' is short for Slot PreScheduling. '+' denotes the *benefit*. '—' represents the *cost* (or *expense*). '%' denotes the *efficiency*.

4.3 Evaluation

In this section, we experimentally evaluate the performance benefit of DynamicMR. We first evaluate the individual impact of each optimization technique of DynamicMR (i.e.,

DHSA, SEPB, Slot PreScheduling in Section 4.3.2, 4.3.3 and 4.3.4, separately). Next, we present the combined performance improvement achieved by DynamicMR in Section 4.3.5. Finally, we compare our DynamicMR with YARN in Section 4.3.6.

4.3.1 Experimental Setup

Benchmark		Input Data	
Name	Description	Data Source	Data Size (GB)
WordCount	computes the occurrence frequency of each word in a document.	wikipedia[12]	10
Sort	sorts the data in the input files in a dictionary order.	wikipedia[12]	20
Grep	finds the matches of a regex in the input files.	wikipedia[12]	30
InvertedIndex	takes a list of documents as input and generates word-to-document indexing.	wikipedia[12]	40
Classification	classifies the input into one of k pre-determined clusters.	movie ratings dataset[12]	20
HistogramMovies	generates a histogram of input data and is a generic tool used in many data analyses.	movie ratings dataset[12]	20
HistogramRatings	generates a histogram of the ratings as opposed to that of the movies based on their average ratings.	movie ratings dataset[12]	20
SequenceCount	generates a count of all unique sets of three consecutive words per document in the input data.	wikipedia[12]	30
SelfJoin	generates association among k+1 fields given the set of k-field associations.	synthetic data[12]	40

Table 4.5: The job information for Purdue MapReduce benchmarks and data sets.

We ran our experiments in a cluster consisting of 10 compute nodes, each with two Intel X5675 CPUs (6 CPU cores per CPU with 3.07 GHz), 24GB memory and 56GB hard disks. We configure one node as master and namenode, and the other 9 nodes as slaves and datanodes. Moreover, we configure 10 map and 2 reduce slots per slave node. The latest version of Hadoop 1.2.1 is chosen in our experiment. We generate our testbed workloads by choosing 9 benchmarks arbitrarily from Purdue MapReduce Benchmarks Suite[39] and using their provided datasets as show in Table 4.5, where the data input sizes are chosen according to the processing capability of our cluster.

4.3.2 Performance Evaluation for DHSA

In this section, we first show the dynamic tasks execution processes for PI-DHSA and PD-DHSA. Then we evaluate and compare the performance improvement by PI-DHSA and PD-DHSA under different slot configuration. Third, we make a discussion on the performance

influence of the arguments of the percentage of map and reduce slots that can be borrowed for our DHSA.

4.3.2.1 Dynamic Tasks Execution Processes for PI-DHSA and PD-DHSA

Figure 4.9: The execution flow for the two DHSAs. There are three pools, with one running job each.

To show different levels of fairness for the dynamic tasks allocation algorithms, PI-DHSA and PD-DHSA, we perform an experiment by considering three pools, each with one job submitted. Figure 4.9 shows the execution flow for the two DHSAs, with 10 sec per time step. The number of running map and reduce tasks for each pool at each time step is recorded. For PI-DHSA, as illustrated in Figure 4.9(a), we can see that, at the beginning, there are only map tasks, with all slots used by map tasks under PI-DHSA. Each pool shares $\frac{1}{3}$ of the total slots (i.e., 36 slots out of 108 slots), until the 5th time step. The map slot demand for Pool 1 begins to shrink and the unused map slots of its share are yielded to Pool 2 and Pool 3 from the 6th time step. Next from 6th to 10th time step, the map tasks from Pool 2 and Pool 3 equally share all map slots and the reduce tasks from Pool 1 possess all reduce slots, based on the type-phase level fairness policy of PI-DHSA (i.e., intra-phase dynamic slot allocation). From 11th to 18th time step, there are some unused map slots from Pool 2 and they are possessed by map tasks from Pool 3 (i.e., intra-phase dynamic slot allocation). Later, there are some unused map slots from Pool 3 and they are used by reduce tasks from Pool 1 and Pool 2 from 22st to 25th time step (i.e., inter-phase dynamic slot allocation).

For PD-DHSA, similar to PI-DHSA at the beginning, each pool obtains $\frac{1}{3}$ of the total slots from the 1th to 5rd time step, as shown in Figure 4.9(b). Some unused map slots from Pool 1 are yielded to Pool 2 and Pool 3 from 6th to the 7th time step. However, from the 8th to 12th, the map tasks from Pool 2 and Pool 3 and the reduce tasks from Pool 1 takes $\frac{1}{3}$ of the total slots, subject to the pool-level fairness policy of PD-DHSA (i.e., intra-pool dynamic slot allocation). Finally, the unused slots from Pool 1 begins to yield to Pool 2 and Pool 3 since 13th time step (i.e., inter-pool dynamic slot allocation).

4.3.2.2 Performance Improvement Comparison

Figure 4.10 presents the performance improvement results in comparison with original Hadoop under various slot configurations, for our proposed DHSA. Note that there are 12 CPU cores per slave node and we assume that one MapReduce slot corresponds to a CPU core. Thereby, we vary the number of map slots per slave node from 1 to 11. Particularly, we define the *speedup* here as the ratio of the execution time of the original Hadoop under 1/11 map/reduce slot configuration per slave node, to the current execution time.

We have the following three observations.

Firstly, the original Hadoop is very sensitive to the map/reduce slot configuration, whereas there is little impact for the map/reduce slot configuration on our DHSA (i.e., the speedup keeps stable under different map/reduce slot configurations). For example, there are about 1.8x performance differences for Sort benchmark in Figure 4.10 (a) between the optimal and worst-case map/reduce slot configurations for the original Hadoop.

To explain the reason behind it, let's take a single job for example. Let N_M and N_R denote the number of map tasks and reduce tasks. Let t_M and t_R denote the execution time for a single map task and reduce task. Let S_M and S_R denote the number of map slots and reduce slots. Moreover, we assume that there is one slot per CPU core and thus the sum of map slots and reduce slots is fixed for a given cluster. Then for the traditional Hadoop cluster, the execution time will be $\lceil \frac{N_M}{S_M} \rceil \cdot t_M + \lceil \frac{N_R}{S_R} \rceil \cdot t_R$. In contrast, it will be $\lceil \frac{N_M}{S_M + S_R} \rceil \cdot t_M + \lceil \frac{N_R}{S_M + S_R} \rceil \cdot t_R$ for our DHSA. Based on the formula, we can see varied performance from the traditional Hadoop under different slot configurations. However, there is little impact on the performance for different slot configurations under DHSA.

Figure 4.10: The performance improvement by DHSA under various slot configuration for MapReduce workloads.

Secondly, compared with the original Hadoop, both PI-DHSA and PD-DHSA can improve the performance of MapReduce jobs significantly, especially under the worst-case map/reduce slot configuration. For example, there are about 2x performance improvement for Sort benchmark under the worst-case configuration (e.g., the x-axis point 1/11 in Figure 4.10(a)), with our proposed DHSA.

Thirdly, the performance improvement is stable and very close to each other for both PI-DHSA and PD-DHSA. The reason is that, although PI-DHSA and PD-DHSA have different fairness concepts (See Section 4.2.2), they follow strictly the same principle of slot utilization maximization, by switching the allocation of the map/reduce slots for map/reduce tasks dynamically.

4.3.2.3 Discussion on the Performance of Different Percentages of Borrowed Map and Reduce Slots

Figure 4.11: The performance results with different percentages of map (or reduce) slots borrowed.

In Section 4.2.2.1, instead of borrowing all unused map (or reduce) slots for overloaded reduce (or map) tasks in our DHSA, we provide users with two configuration arguments *percentageOfBorrowedMapSlots* and *percentageOfBorrowedReduceSlots* to limit the amount of borrowed map/reduce slots, and ensure that tasks at the map/reduce phase are not starved. It is meaningful and important when users want to reserve some unused slots for incoming tasks, instead of lending all of them to other phases or pools. To show its impact on the performance, we perform an experiment with *sort* benchmark (320 map tasks and 200 reduce tasks) by varying values of arguments.

Let Pec_M and Pec_R denote *percentageOfBorrowedMapSlots* and *percentageOfBorrowedReduceSlots* respectively. Figure 4.11 presents the performance results under varied

argument configurations. All speedup results are calculated with respect to the case when $Pec_M = 0$ and $Pec_R = 0$. We consider four cases: (1). Vary the value of Pec_R from 0 to 100 while fix $Pec_M = 0$; (2). Vary the value of Pec_R while set $Pec_M = 100$; (3). Vary the value of Pec_M while fix $Pec_R = 0$; (4). Vary the value of Pec_M while fix $Pec_R = 100$. We can see that, the performance improves by increasing either Pec_M or Pec_R . Particularly, there is a significant performance improvement (i.e., approximate 29%) when we fix the value of Pec_R and increase the value of Pec_M . It is because there are plenty of reduce tasks (e.g., 200 reduce tasks) but only 18 reduce slots. Thus increasing the percentage value of map slots (Pec_M) that can be borrowed would let more reduce tasks be scheduled using borrowed map slots, reducing the number of computation waves of reduce tasks and improving the utilization as well as performance of the Hadoop cluster.

4.3.3 Speculative Execution Control for Performance

(a) The performance improvement under different percentages. (b) The detailed job execution time for the workload of 5 jobs.

Figure 4.12: The performance results with SEPB.

Recall in Section 4.2.3, we stated that speculative task execution can overcome the problem of straggler (i.e., the slow-running task) for a job, but it is at the cost of cluster utilization. We define a user's configurable variable *percentageOfJobsCheckedForPendingTasks* to determine the time to schedule speculative tasks. To validate the effectiveness of our dynamic speculative execution control policy, we perform an experiment with 5 jobs, 10 jobs and 20 jobs by varying the values of *percentageOfJobsCheckedForPendingTasks*.

Note that LATE[102] has been implemented in Hadoop 1.2.1. Figure 4.12 presents the performance results with SEPB in comparison to LATE. All speedups are computed with respect to the case that *percentageOfJobsCheckedForPendingTasks* is equal to zero. We have the following findings:

First, SEPB can improve the performance of Hadoop from 3% ~ 10%, shown in Figure 4.12(a). As the value of *percentageOfJobsCheckedForPendingTasks* increases, the performance improvement tends to be larger and the optimal configurations could be distinct for different workloads. For example, the optimal configuration for 5 jobs is 80%, but for 10 jobs is 100%. The reason is that, large value of *percentageOfJobsCheckedForPendingTasks* will let more numbers of jobs be checked for pending tasks before considering speculative execution for each slot allocation, i.e., It is more likely to allocate a slot to a pending task first, rather than a speculative task, which benefits more for the whole jobs. However, large value of *percentageOfJobsCheckedForPendingTasks* will delay the speculative execution for straggled jobs, hurting their performance. For some workloads, too large value of *percentageOfJobsCheckedForPendingTasks* will degrade the performance for straggled jobs a lot and in turn affect the overall jobs, explaining why the optimal configuration is not always 100%. We recommend users to configure *percentageOfJobsCheckedForPendingTasks* at 60% ~ 100% for their workloads.

Second, there is a performance tradeoff between an individual job and the whole jobs with SEPB. We show a case for the workload of 5 jobs when setting *percentageOfJobsCheckedForPendingTasks* to be 0 and 100%, respectively. As results shown in Figure 4.12(b), Job 2 and 4 are negative affected due to the constrain on speculative execution from SEPB, whereas it favors the performance for whole jobs (i.e., the maximum execution time of jobs).

4.3.4 Data Locality Improvement Evaluation for Slot PreScheduling

To test the effect of Slot PreScheduling on data locality improvement, we ran MapReduce jobs with 16, 32, and 160 map tasks on the Hadoop cluster. We compare fair sharing results with and without Slot PreScheduling under the default HFS. It is worth mentioning that Delay Scheduler has been added to the default HFS for the traditional Hadoop and keeps

working always. Therefore, our work turns to be the comparison between the case with Delay Scheduler only and the case with Delay Scheduler plus Slot PreScheduling.

Figure 4.13: The data locality improvement for Slot PreScheduling.

Figure 4.13 shows the data locality results with and without Slot PreScheduling for numerous benchmarks. With Slot PreScheduling, there are about 2% ~ 25% locality improvement on top of Delay Scheduler for *Sort* benchmark.

Finally, Figure 4.14 presents the corresponding performance results benefiting from the data locality improvement made by Slot PreScheduling. There are about 2% ~ 25% performance improvement with respect to the original Hadoop for the aforementioned 9 benchmarks respectively.

Figure 4.14: The performance improvement under Slot PreScheduling.

(a) A single MapReduce job

(b) MapReduce workloads with multiple jobs

Figure 4.15: The performance improvement with our DynamicMR system for MapReduce workloads.

4.3.5 Performance Improvement Evaluation for DynamicMR

In this section, we evaluate DynamicMR system by enabling all its three sub-schedulers so that they can work corporately to maximize the performance as much as possible. For DHSA part, we arbitrarily choose PI-DHSA, noting that PI-DHSA and PD-DHSA have very similar performance improvement (See Section 4.3.2.2). For the original Hadoop, we choose 6/2 as its map/reduce slot configuration for each tasktracker, according to evaluation results in Figure 4.10. We aim to compare the performance for DynamicMR with the original Hadoop under the optimized map/reduce slot configuration.

Figure 4.15 presents the evaluation results for a single MapReduce job as well as MapReduce workloads consisting of multiple jobs. Particularly, for multiple jobs, we consider 5

jobs, 10 jobs, 20 jobs, and 30 jobs under a batch submission, i.e., all jobs submitted at the same time. All speedups are calculated with respect to the original Hadoop. We can see that, even under the optimized map/reduce slot configuration for the original Hadoop, our DynamicMR system can still further improve the performance of MapReduce jobs significantly, i.e., there are about 46% ~ 115% for a single job and 49% ~ 112% for MapReduce workloads with multiple jobs.

Moreover, we also implement our DynamicMR for Hadoop FIFO scheduler. To validate the effectiveness of our DynamicMR, we perform experiments with the aforementioned MapReduce workloads. The results are shown in Figure 4.16. It illustrates that, our DynamicMR system can improve the performance of Hadoop jobs significantly under FIFO scheduler as well.

Figure 4.16: The performance improvement with our DynamicMR system for MapReduce workloads under Hadoop FIFO scheduler.

4.3.6 Performance Comparison With YARN

In YARN, there is no more concept of 'slot'. Instead, it proposes a concept of 'container' consisting of a certain amount of resources (e.g., memory) that both map and reduce tasks can run on. It is claimed that it can overcome the utilization problem of static slot-based approach. In this section, we perform an experimental comparison between YARN and our DynamicMR.

Figure 4.17: The comparison results between YARN and DynamicMR for MapReduce workloads.

To make it comparable, in our argument settings of YARN, we configure the allocated memory resources for each container carefully so that the number of containers in each slave node is equal to the number of 'slot' in Hadoop MRv1. We also check other same arguments (e.g. *mapreduce.job.reduce.slowstart.completedmaps*) to ensure that they have the same configured value for YARN and Hadoop MRv1.

Figure 4.17 shows the compared performance results of speedup with respect to YARN. For single MapReduce jobs in Figure 4.17 (a), we can not claim that which one is better than the other absolutely. For example, Our DynamicMR outperforms YARN for benchmarks Sort, SelfJoin and Classification, whereas YARN is better than DynamicMR for other remaining benchmarks. This is because with a single job, there is no difference in resource utilization optimization mechanism between YARN and DynamicMR (i.e., both of them use all resources for map tasks at the map-phase first and then utilize all resources for reduce tasks at the reduce-phase).

However, for multiple jobs, we can see in Figure 4.17 (b) that our DynamicMR is better than YARN, especially when the number of jobs is large. The reason is due to the network contention mainly from reduce tasks caused in their shuffle phase. Given a certain number of resources, it is obvious that the performance for the case with a ratio control of concurrently running map and reduce tasks is better than without control. Because without control, it easily occurs that there are too many reduce tasks running, causing the network to be a bottleneck seriously.

For YARN, both map and reduce tasks can run on any idle container. There is no control mechanism for the ratio of resource allocation between map and reduce tasks. It means that when there are pending reduce tasks, the idle container will be most likely possessed by them. In contrast, our DynamicMR follows the traditional slot-based model. In contrast to the ‘hard’ constrain of slot allocation that map slots have to be allocated to map tasks and reduce tasks should be dispatched to reduce tasks, we propose a ‘soft’ constrain of slot allocation to allow that map slot can be allocated to reduce task and vice versa. But whenever there are pending map tasks, the map slot should be given to map tasks first, and the rule is similar for reduce tasks. It means that, the traditional way of static map/reduce slot configuration for the ratio control of running map/reduce tasks still works for DynamicMR. In comparison to YARN which maximizes the resource utilization only, our DynamicMR can maximize the slot resource utilization and meanwhile dynamically control the ratio of running map/reduce tasks via map/reduce slot configuration.

To validate our hypothesis, we make a throughput test for data shuffling of reduce tasks over time by running a MapReduce workloads of 5 jobs. Figure 4.18 illustrates the shuffle data throughput for YARN and DynamicMR. The larger throughput indicates that there are more reduce tasks performing data shuffling. We can see that the data throughput for YARN fluctuates greatly over time and its peak value is much higher than DynamicMR, demonstrating the correctness of our clarification.

Figure 4.18: The network throughput comparison between YARN and DynamicMR for data shuffling over time. Each *time step* are 2 seconds.

4.4 Comparison between DynamicMR and Job Ordering Optimization

In this thesis, we so far provide two approaches to improve the performance for slot-based Hadoop MRv1 cluster. One is the job ordering optimization approach introduced in Chapter 3. The other is DynamicMR system proposed in this chapter. In this section, we make a comparison between the two approaches from the following aspects:

Hadoop Source Code Modification. For the job ordering optimization, it does not need any modification for the Hadoop source code. It is achieved by optimizing the job submission order from the workload aspect. In contrast, DynamicMR is implemented by modifying the Hadoop source code.

Assumption Requirement. There are some assumptions/information needed for job ordering optimization. First, the execution time for map/reduce tasks per job should be known in advance. Second, the number of map/reduce tasks for each job should also be provided. Third, the arriving time for each job should be given for online MapReduce workloads. Fourth, it assumes no data dependency between jobs. In contrast, there is no any assumption for DynamicMR here.

Generality. For job ordering approaches in Chapter 3, it can only support a limited number of applications/workloads (e.g., production workloads that run periodically for new data processing) due to its assumption. For DynamicMR, it can support any type of MapReduce workloads (e.g., single MapReduce job, multiple independent/dependent MapReduce jobs).

Optimization Degree. DynamicMR can achieve better and more substantial performance improvement than the job ordering approach. The explanation is that, the job ordering optimization achieves the performance optimization by adjusting the job submission order so that the overlapping computation between map tasks and reduce tasks can be maximized for batch jobs. However, it is still vulnerable to the map/reduce slot configuration and the characteristic of MapReduce workloads. There is still some idle slots during the computation even for the optimal job order, as illustrated in Figure 3.1(b) of Chapter 3. Moreover, the job ordering cannot work for a single MapReduce job case.

In contrast, DynamicMR can maximize slot utilisation during the computation, attributing to its dynamic allocation mechanism that enables map slots to be used dynamically for reduce tasks when the number of map tasks is small, and vice versa. Both the performance for the batch jobs case and the single job case can be optimized with DynamicMR.

Supported Schedulers. The job ordering optimization can only work for FIFO scheduler. In contrast, although we focus on the Fair Scheduler for DynamicMR in this chapter, it indeed can support FIFO scheduler as well.

4.5 Summary

This chapter proposes a general framework called DynamicMR (by modifying the source code of Hadoop) to improve the performance of Hadoop MRv1 cluster. Compared with the job ordering approaches in Chapter 3, DynamicMR has a better performance improvement for Hadoop cluster, since 1). DynamicMR does not need any assumption or requires any prior-knowledge; 2). DynamicMR can be used for any kinds of MapReduce jobs (e.g., independent or dependent ones). All of these are attributed to its three step-by-step optimization techniques, namely, DHSA, SEPB, Slot PreScheduling. DHSA is responsible for the slot utilization maximization by allocating map (or reduce) slots to map and reduce tasks dynamically. Two types of DHSA are presented, namely, PI-DHSA and PD-DHSA, based on different levels of fairness. Users can choose either of them accordingly. In contrast, SEPB identifies the slot inefficiency problem of speculative execution. It balances the performance tradeoff between a single job and a batch of jobs dynamically. Slot PreScheduling improves the performance by maximizing its data locality. The three step-by-step optimization processes enable DynamicMR to optimize the performance of MapReduce workloads substantially. Our experimental results show that DynamicMR can improve the performance of the Hadoop system significantly (i.e., 46% ~ 115% for single jobs and 49% ~ 112% for multiple jobs). Moreover, we also have a comparison with YARN. The experiments show that, 1) for single jobs, the result is inconclusive, 2) for multiple jobs, DynamicMR consistently outperforms YARN by about 2% ~ 9%.

Finally, the DynamicMR source code is publicly available at <http://sourceforge.net/projects/dynamicmr/> or <https://issues.apache.org/jira/browse/MAPREDUCE-5643>.

Chapter 5

LTARN: A Long-Term YARN Fair Scheduler for Pay-as-you-go Computing Systems

5.1 Introduction

We have considered the *performance* optimization for MapReduce workloads in Chapter 3 and Chapter 4. This chapter, in contrast, attempts to optimize the *fairness*. Particularly, we put focus on pay-as-you-go computing environments, such as super-computing and cloud computing (e.g., Amazon EC2), where users only need to pay for resources based on their usages, rather than have to purchase physical machines. However, we observe that, resource sharing policies used in many existing frameworks such as Hadoop and YARN, are *MemoryLess Resource Fairness (MLRF)*, which are *not* suitable for pay-as-you-use computing. We will show that their *memoryless* feature can cause the following problems: 1). users can get benefits by falsely reporting their demands to the system; 2). users in pay-as-you-use computing environment might not be able to get the total amount of resources they are entitled to based on the amount paid; 3). since there is no benefit for releasing resources to others in *memoryless* allocation, selfish users may possess unused idle resources by submitting cost-inefficient workloads. This chapter attempts to address these problems.

The chapter is organized as follows. Section 5.2 presents several payment-oriented resource allocation properties, and defines the notion of fairness for pay-as-you-use computing. Section 5.3 lists three severe problems for existing memoryless resource fairness. To solve the problems, Section 5.4 proposes a long-term resource fairness (LTRF). Section 5.5

implements it in YARN by developing a long-term YARN fair scheduler called *LTYARN*. Section 5.6 evaluates LTYARN experimentally.

5.2 Allocation Properties and Fairness Definition

This section gives formal definitions for resource allocation properties and fairness, which are used to assess fair policies in the following sections.

5.2.1 Resource Allocation Properties

We present several resource allocation properties that are essential and desirable for any cloud-oriented resource allocation policy.

- *Sharing Incentive*: Each user should be better off sharing the resources via group-buying with others, than exclusively buying and using the resources individually[42].
- *Cost-Efficient Workload Incentive*: In pay-as-you-go computing environments, since the resources are not free, we should encourage users to submit cost-efficient workloads¹, instead of unnecessary workloads. To achieve that, it needs a mechanism that benefits the user who yields unused resources to others when not needed. Otherwise, she may be selfish and possess all unneeded resources under her share (e.g., running some dirty tasks in a shared computing environment)[90].
- *Resource-as-you-pay Fairness*: The resource a user gained should be proportional to her payment. This property is important as it is a SLA guarantee to users[90].
- *Truthfulness*: Users should not benefit from providing false information about their resource demand[95]. This property is compatible with sharing incentive and resource-as-you-pay fairness, since no user can obtain more resources by lying.
- *Pareto Efficiency*: The resource gain by one user will be at the expense of other users. It should not be possible for a user to get more resources without decreasing the resource of at least one user[42]. This property can ensure the system resource utilization to be maximized.

¹Cost-efficient workload: refers to the *non-trivial* workload that users really want to submit and run in pay-as-you-computing system

5.2.2 Fairness Definition

Resource-as-you-pay fairness and sharing incentive property are two key fundamental metrics for any fairness policy in a shared pay-as-you-go computing system. Due to the varied resource demands and resource preemption in the shared environment, the total resources a user obtained are undermined. Generally, every user wants to get more resources or at least the same amount of resources in a shared computing system than if they have bought them individually. We call it *fair* for a user (i.e., sharing benefit) when this can be achieved. In contrast, it is also possible for the total resources a user received are less than what she would get if she has bought and used the resources individually, which we call *unfair* (i.e., sharing loss). To ensure resource-as-you-pay fairness and the maximization of sharing incentive property in the shared system, it is important to minimize *sharing loss* firstly and then maximize *sharing benefit*.

In the remainder of this chapter, we refer to the *total* resources as *accumulated* resources below. Let $g_i(t)$ be the current allocated resources for the i^{th} user at time t . Let $f_i(t)$ denote the *accumulated* resources for the i^{th} user at time t . Thus,

$$f_i(t) = \int_0^t g_i(t) dt. \quad (5.1)$$

Let $d_i(t)$ and $S_i(t)$ denote the current demand and current resource share for the i^{th} user at time t , respectively. Given the total resource capacity R of the system and the shared weight w_i for the i^{th} user, there is

$$S_i(t) = R \cdot w_i / \sum_{k=1}^n w_k. \quad (5.2)$$

The fairness degree $\beta_i(t)$ for the i^{th} user at time t is defined as the normalization result of the amount of resources a user obtained in a shared environment with respect to the non-shared environment, i.e.,

$$\beta_i(t) = \frac{AllocationResultWithSharing}{AllocationResultWithoutSharing} = \frac{\int_0^t g_i(t) dt}{\int_0^t \min \{d_i(t), S_i(t)\} dt}. \quad (5.3)$$

$\beta_i(t) \geq 1$ implies the absolute resource fairness for the i^{th} user at time t . In contrast, $\beta_i(t) < 1$ indicates *unfair*. We can easily see that for a user i in a non-shared partition of the system, it always holds $\beta_i(t) = 1$, since it has $g_i(t) = \min \{d_i(t), S_i(t)\}$ at any time t in this scenario. To measure the gain/loss, in term of resource, to the system of resource sharing with a fair policy compared to without sharing, we propose two new metrics *sharing benefit degree* and *sharing loss degree*. Let $\Psi(t)$ be *sharing benefit degree*, as a sum of all $(\beta_i(t) - 1)$ subject to $\beta_i(t) \geq 1$, i.e.,

$$\Psi(t) = \sum_{i=1}^n \max \{\beta_i(t) - 1, 0\}. \quad (5.4)$$

and let $\Omega(t)$ denote *sharing loss degree*, as a sum of all $(\beta_i(t) - 1)$ subject to $\beta_i(t) < 1$, i.e.,

$$\Omega(t) = \sum_{i=1}^n \min \{\beta_i(t) - 1, 0\}. \quad (5.5)$$

Thereby, it always holds that $\Psi(t) \geq 0 \geq \Omega(t)$. Moreover, in a non-shared environment, it always holds $\Psi(t) = \Omega(t) = 0$, indicating that there are neither sharing benefit nor sharing loss. In contrast, in a shared pay-as-you-go computing system, either of them could be nonzero. For a good fair policy, it should be able to maximize $\Omega(t)$ first (e.g., $\Omega(t) \rightarrow 0$) and next try to maximize $\Psi(t)$ as much as possible. Finally, we can use these two metrics to measure and compare the quality for different fair policies.

5.3 MemoryLess Resource Fairness (MLRF)

(*Weighted*) *Max-Min Fairness* is one of the most popular fair allocation policies[42]. It maximizes the minimum resource allocation obtained by a user in a shared computing system. It has been widely used in many popular high performance computing frameworks such as Hadoop[5], YARN[91], Mesos[57], Dryad[59] and Choosy[43]. Unfortunately, we observe that the fair polices implemented in these systems are all *memoryless*, i.e., allocating resources fairly at the current moment without considering history information. We refer those schedulers with *MemoryLess Resource Fairness (MLRF)*. It is *not* suitable for such pay-as-you-go computing system because of the following problems:

Cost-Inefficient Workload Submission Problem. In a pay-as-you-go computing system, we should have a mechanism to encourage group members to submit cost-efficient workload, instead of unnecessary workloads. For *MLRF*, there is an implicit assumption that all users are not selfish and honest towards their requested resource demands, which is however often not true in real world. It can cause cost-inefficient workload problem with *MLRF*. Consider two users A and B sharing a system for example. Let D_A and D_B be the *true* workload demand for A and B at time t_0 , respectively. Assume that D_A is less than its share² while D_B is larger than its share. In that case, it is possible that A is selfish and will try to use up all of her share of the resources by running unnecessary tasks/jobs (e.g., running some duplicated tasks of the experimental workloads for double checking) to deny others of using the resource, causing the inefficiency problem of running cost-inefficient workloads and breaking the sharing incentive (See definition in Section 5.2.1) property at the same time.

Untruthfulness Problem. It is important for a shared system to have a policy to ensure that no group member can get benefits by lying (See *Truthfulness* in Section 5.2.1). We argue that *MLRF* cannot satisfy this property. Consider a system consisting of three users A , B , and C . Assume A and C are honest while B is not honest. It could happen at a time that both the *true* demands of A and B are less than their own shares while C 's *true* demand exceeds its share. In that case, A yields her unused resources to others honestly. But B will provide *false* information about her demand (says, far larger than her share) and compete with C for unused resources from A , benefiting B and hence violating truthfulness. Moreover, it will break the sharing incentive property as well if all other users do the same thing like B .

Resource-as-you-pay Unfairness Problem. For group-buying resources, we should ensure that the total resources received by each member are proportional to their monetary contribution (See *Services-as-you-pay Fairness* in Section 5.2.1). Due to the varied resource demands (e.g., workflows) for a user at different time, our work shows that *MLRF* fails at this point. Consider two users A and B . At time t_0 , it could happen that the demand D_A

²By default, we refer to the *current* share at the designated time (e.g., t_0), rather than the *total* share accumulated over time.

is less than its allocated share of the resources and hence its extra unused resource will be possessed by user B (i.e., lend to B) according to the work conserving property of *MLRF*. Next at time t_1 , assume that user A 's demand D_A becomes larger than its share. With *MLRF*, however, A can only use her current share (i.e., cannot claim back the resources that she has given to B at time t_0 , if D_B is larger than its share, due to *memoryless*). If this scenario occurs frequently, it will be unfair for A to get the amount of resources she should have obtained from a long-term view. (See a motivation example in Section 5.4).

5.4 Long-Term Resource Fairness(LTRF)

In this section, we first give a motivating example to show that MemoryLess Resource Fairness (MLRF) is *not* suitable for pay-as-you-go computing system. Next we propose Long-Term Resource Fairness (LTRF), a payment-oriented allocation policy to address it and meet the desired properties in the previous section.

Motivating Example. Consider a shared computing system with 200 resources (e.g., 200GB RAM) and two users A, B with equal share of 100GB each. As shown in Table 5.1, suppose that the new requested demands at time t_1, t_2, t_3, t_4 for user A are 40, 80, 160, 120, and for user B are 200, 120, 100, 100, respectively. With MLRF, we see in Table 5.1(a) that, at t_1 , the total demand and allocation for A are both 40. It lends 60 unused resources to B and thus 160 allocations for B . The scenario is similar at t_2 . Next at t_3 and t_4 , the total demand for A becomes 160 and 180, bigger than its share of 100. However, it can only get 100 allocations based on MLRF, being *unfair* for A since it makes the total allocations for A and B become $320(= 40 + 80 + 100 + 100)$ and $480(= 160 + 120 + 100 + 100)$ at time t_4 , respectively. Instead, if we adopt LTRF, as shown in Table 5.1(b), the total allocations for A and B at t_4 will finally be the same (e.g., 400), being *fair* for A and B .

LTRF Algorithm. The pseudo-code for LTRF is shown in Algorithm 10. It considers the fairness of total allocated resources consumed by each user, instead of current allocated resources. The core idea is based on the '*loan(lending) agreement*'[8] with free interest. That is, a user yields her unused resources to others as a *lend* manner at a time. When she needs resources at a later point in time, the user should get the resources back from others

	User A					User B				
	Demand		Allocation		Preempt	Demand		Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	40	40	40	40	-60	200	200	160	160	+60
t_2	80	80	80	120	-20	120	160	120	280	+20
t_3	160	160	100	220	0	100	140	100	380	0
t_4	120	180	100	320	0	100	140	100	480	0

(a) Allocation results based on *MLRF*. *Total Demand* refers to the sum of the new demand and accumulated remaining demand in previous time.

	User A					User B				
	Demand		Allocation		Preempt	Demand		Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	40	40	40	40	-60	200	200	160	160	+60
t_2	80	80	80	120	-20	120	160	120	280	+20
t_3	160	160	160	280	+60	100	140	40	320	-60
t_4	120	120	120	400	+20	100	200	80	400	-20

(b) Allocation results based on *LTRF*.

	User A					User B				
	Demand		Counted Allocation		Preempt	Demand		Counted Allocation		Preempt
	New	Total	Current	Total		New	Total	Current	Total	
t_1	40	40	40	40	-60	200	200	130	130	+60
t_2	80	80	80	120	-20	120	160	110	240	+20
t_3	160	160	130	250	+60	100	140	40	280	-60
t_4	120	120	110	360	+20	100	200	80	360	-20

(c) Counted allocation results under *discount*-based approach of *LTRF*. There is a discount (e.g., 50%) for the extra unused resources, to incentivize users to preempt resources actively for system utilization maximization. In this example, although the *counted* allocations for A and B are 360, their real allocations are both 400, which is the same as Table 5.1(b).

Table 5.1: A comparison example of *MemoryLess Resource Fairness (MLRF)* and *Long-Term Resource Fairness (LTRF)* in a shared computing system with 200 computing resources for two users A and B.

that she yielded before (i.e., *return* manner). In our previous two-user example with LTRF in Table 5.1(b), user A first lends her unused resources of 60, 20 to user B at time t_1 and t_2 , respectively. However, at t_3 and t_4 , she has a large demand and then collects all 80 extra resources back from B that she lent before, making *fair* between A and B .

However, due to the *lending agreement* of LTRF, in practice, when A yields her unused resources at t_1 and t_2 , B might not want to acquire the extra unused resources from A immediately if her application's deadline is not so tight. In that case, the total allocations for A and B will be $320(= 40 + 80 + 100 + 100)$ and $400(= 100 + 100 + 100 + 100)$ at time t_4 , causing the inefficiency problem for the system utilization. To solve this problem, we propose a *discount*-based approach. The basic idea is to sell the unused resources to other users in the group at a discount rate (e.g., 50%) of current rate. It will incentivize B to preempt extra unused resources from A , since it is *cheaper* than its own share of resources. Alternatively, Net Present Value (NPV)[58] is a widely used discount model in finance market. It is defined as the sum of the present values (PVs) of incoming and outgoing cash flows over a period of time. In comparison, the discount rate of our proposed discount approach is static and not degraded or changed over time, whereas NPV does not. There is a key assumption for NPV model that the discount rate is decreasing (i.e. changing) over time. That is, if for example there are a time series of identical cash flows, the cash flow in the present is the most valuable, with each future cash flow becoming less valuable than the previous cash flow over time. However, with our discount model, for A , it is not a lost from a long-term point of view, as it can get the same *discount* on the resource counting for the preempted resources from B back later.

Table 5.1(c) demonstrates this point. It shows the counted total resource allocation for each user over time by discounting the possessed extra unused resources. At time t_1 , A yields her 60 unused resources to B and B 's counted total resources are $130(= 100 + 60 \cdot 50\%)$ instead of $160(= 100 + 30)$. Similarly for A at t_3 , it preempts 60 resources from B and its counted total resources are $130(= 100 + 60 \cdot 50\%)$. Both of them are *fair* at time t_4 with the equal counted total resource allocation of 360.

In contrast, if with the NPV discount model, it is most likely that user A cannot get the same amount of resources back from user B due to the different discounting rate over time

for the same amount of resources, which is not what we want to achieve.

Algorithm 10 LTRF pseudo-code.

```

1:  $\bar{R}$ : total resources available in the system.
2:  $\bar{R} = (\bar{R}_1, \dots, \bar{R}_n)$ : current allocated resources.  $\bar{R}_i$  denotes the current allocated resources for user  $i$ .
3:  $U = (u_1, \dots, u_n)$ : total used resources, initially 0.  $u_i$  denotes the total resource consumed by user  $i$ .
4:  $W = (w_1, \dots, w_n)$ : weighted share.  $w_i$  denotes the weight for user  $i$ .

5: while there are pending tasks do
6:   Choose user  $i$  with the smallest total weighted resources of  $u_i/w_i$ .
7:    $d_i \leftarrow$  the next task resource demand for user  $i$ .
8:   if  $\bar{R}_i + d_i \leq \bar{R}$  then
9:      $\bar{R}_i \leftarrow \bar{R}_i + d_i$ . ▷ Update current allocated resources.
10:    Update the total resource usage  $u_i$  for user  $i$ .
11:    Allocate resource to user  $i$ .
12:   else ▷ The system is fully utilized.
13:     Wait until there is a released resource  $r_i$  from user  $i$ .
14:      $\bar{R}_i \leftarrow \bar{R}_i - r_i$ . ▷ Update current allocated resources.
15:   end if
16: end while

```

5.4.1 Property Analysis for LTRF

Theorem 5.4 *LTRF satisfies the sharing incentive property.*

Proof Consider a shared pay-as-you-go computing system of R resources group-bought by n users with equal share (or monetary cost) over t time period. When pursuing resources individually with the same amount of money, 1). the amount of resources R_1 a user can receive is less than $\frac{R}{n}$, as group-buying has discount over personal buying; 2). Under R_1 resources, she can get at most $t \cdot R_1$ resources, smaller than $t \cdot \frac{R}{n}$. In contrast, with group-buying and fair allocation with LTRF, a user can get at least $t \cdot \frac{R}{n}$ resources. Thus LTRF satisfies sharing incentive property. □

Theorem 5.5 (*Cost-Efficient Workload Incentive*) *Any user who submits cost-efficient workloads to the shared pay-as-you-go computing system could get benefits under LTRF.*

Proof Recall that LTRF focuses on the fairness of total resources over the entire time period with *lending agreement*. When a user's resource demand is less than its current share, she

can *lend* unneeded resources out. Later when she needs more resources in the future, she can get extra amount of resources back from others. Conversely, if she submits lots of cost-inefficient (e.g., dirty) workloads to the system when her *true* demand is less than her share, she will loose chance to get more extra resources, especially when she has lots of important and urgent workloads to compute later. Hence, LTRF satisfies cost-efficient workload incentive property. $\square$

Theorem 5.6 *LTRF is resource-as-you-pay fairness in a group-buying shared computing system.*

Proof Each user in a *shared* computing system has the right to enjoy at least the amount of resources she has paid in a non-shared environment. One key factor that affects resource-as-you-pay fairness is the varied user's demands at different time (i.e., *unbalanced* workload which can be either less or larger than her current share). LTRF overcomes the unbalanced workload problem by considering the fairness at the level of total allocated resources and following *lending agreement*. It adjusts the current allocation of resources to each user dynamically according to the total resource consumed by the user and current demand, making sure that the total resources a user received are *fair* to others. Thus LTRF is resource-as-you-pay fairness. $\square$

Theorem 5.7 *LTRF satisfies truthfulness property.*

Proof Theorem 5.5 has demonstrated that LTRF satisfies cost-efficient workload incentive property that can encourage users to yield unused resources. On the other hand, it is possible that an overloaded user lies about her true demands to get more allocated resources in preemption with others. However, due to *lending agreement* requirement under LTRF, the user needs to "pay back" at a later point in time. Thus, lying cannot benefits her at all and our proof is complete. $\square$

Theorem 5.8 *LTRF satisfies pareto efficiency property.*

Proof Recall in our LTRF algorithm, we propose a *discount*-based approach to incentivize users to use idle resources. It indicates that the utilization of system is fully maximized whenever there are pending tasks. That is, it is impossible for a user to get more resources without disadvantaging other users and thereby our proof is complete. . $\square$

Finally, Table 5.2 summarizes the properties that are satisfied by MLRF and LTRF, respectively. We can see that MLRF is *not* suitable for pay-as-you-go computing system due to its lack of support for three important desired properties. In contrast, LTRF, is *suitable* for pay-as-you-go computing system.

Property	Allocation Policy	
	MLRF	LTRF
Sharing Incentive	✓	✓
Cost-Efficient Workload Incentive		✓
Resource-as-you-pay Fairness		✓
Truthfulness		✓
Pareto Efficiency	✓	✓

Table 5.2: List of properties for MLRF and LTRF.

5.5 LTYARN: A Long-Term YARN Fair Scheduler

YARN is a most popular resource management and job processing system, and has been viewed as a distributed operating system. In this section, we consider implementing LTRF on YARN. We propose a long-term YARN fair scheduler called *LTYARN*, by generalizing the default *instant* max-min fairness (Section 5.5.1). Based on that, we design and implement our LTYARN for YARN (Section 5.5.2).

5.5.1 Long-Term Max-Min Fairness

In this subsection, we present our long-term max-min fairness method. We will first highlight the challenges in achieving long term fairness and describe our solutions to them. Next we will propose our long-term max-min fairness model for LTYARN.

5.5.1.1 Challenges and Approaches

Our long-term max-min fairness policy is based on the *accumulated* resources. When estimating the *accumulated* resources for a task, we need to know the capacity and demand of its requested resources and the execution time it takes. However, there are several challenges for *online* applications on that as follows,

1. the execution time of tasks for each application are often different and unknown in advance.
2. the arriving time for each application can be arbitrary and unknown in advance.
3. the computing resources (e.g., CPU powers) can be heterogeneous in a heterogeneous cluster, and the resource demand (e.g., memory size) for each task can be different.

To deal with the above mentioned challenging issues, we provide several methods below,

Time Quantum-based Approach. It is an approximation approach to deal with the first challenging problem. It gives a concept of *assumed execution time*, initialized with a time quantum, to represent the task execution time. It can be adjusted dynamically to make it close to the real execution time.

The details of our approach are that, we first initialize the *assumed execution time* to be zero for any *pending* task. When a task starts running, we give a time quantum threshold for its *assumed execution time*. For each running task, when its running time exceeds the *assumed execution time*, the *assumed execution time* is updated to the running time. In contrast, for any finished task, its *assumed execution time* is updated to its running time, no matter it is larger or smaller than the time threshold.

Wall Clock-based Approach. It concerns with the second challenging problem of '*online*' arriving. Note for the *offline* case where all applications arrive at the same time, it would be simple to adopt the accumulated resources *consumed* by each application directly for fairness control. However, for the *online* case, different applications may arrive at different time. It would be no longer suitable to use the accumulated *consumed* resources as a measure to control the fair share. The explanation is that, from the system's (e.g., global-level) perspective, in order to improve its resource utilization, it often follows the idiom

that ‘*the early bird gets the worm*’ (we call it *Early Bird Privilege* next) to incentivize users to submit their applications as early as possible. To achieve that, one solution is to give a penalty for late arriving application, by only starting to consider (or memorize) the fair share of resources from its arriving time. Moreover, it’s worth mentioning that our fairness model is on the basis of max-min fairness algorithm[10]. Technically, to implement it, there is a need to top-up a resource cost, named as *Pseudo Accumulated Resources (PAR)*, such that the fair scheduler will not favor the late arriving application because of its later arriving. Thus, in contrast to *offline* application whose accumulated resources can be directly set to its accumulated *consumed* resources as expressed by Formula (5.1) implicitly, the accumulated resources for each *online* application should include both its *PAR* and accumulated consumed resources. That is, for the online application, Formula (5.1) should be modified as,

$$f_i(t) = \int_0^t g_i(t) dt + \phi_i(t). \quad (5.6)$$

where $\phi_i(t)$ denotes the *PAR watched* at time t by the application i . Moreover, by taking into account the *discount*-based approach for extra unused resources proposed by Algorithm 10 of LTRF in Section 5.4, we have the *counted* current allocated resource $g'_i(t)$ as follows:

$$g'_i(t) = \min\{g_i(t), S_i(t)\} + \max\{g_i(t) - S_i(t), 0\} \cdot \eta. \quad (5.7)$$

where $\eta(0 \leq \eta \leq 1)$ denotes the discount rate. Hence Formula (5.6) should be modified as,

$$f_i(t) = \int_0^t g'_i(t) dt + \phi_i(t). \quad (5.8)$$

We call this method *Wall Clock-based Approach*, where the *Wall Clock* refers to a time period before the arriving of an application, as illustrated in Figure 5.1 (a).

Weighted Resource based Approach. It targets at the third challenging problem. That is, each resource is assigned a weight based on its computing capacity. For example, the

CPU resource can be weighted based on its clock frequency. Thereby, for the i^{th} application,

$$g_i(t) = \sum_{j \in \tau_i(t)} \theta_{i,j} \cdot \delta_{i,j} \cdot \alpha_{i,j}(t). \quad (5.9)$$

where $\tau_i(t)$ denotes the set of tasks from the i^{th} application that are allocated with resources at the time t . $\theta_{i,j}$ and $\delta_{i,j}$ denote the resource demand (e.g., the size of vcore or memory) and weight for the j^{th} task of the i^{th} application, respectively. $\alpha_{i,j}(t)$ represents the *assumed execution time* for the j^{th} task of the i^{th} application at time t .

5.5.1.2 Long-Term Max-Min Fairness Model

Figure 5.1: The long-term max-min fairness models for *LTYARN*. For an application, *Active Period* refers to the time interval when it has pending/running tasks available. Otherwise, it belongs to *Non-active Period*. *Wall Clock* refers to a time period before the arriving of an application with respect to the starting time of the current round.

This subsection proposes long-term max-min fairness model for *LTYARN*. Since *YARN* is a hierarchical tree structures of multi-level fairness, the following part considers the bottom-level (i.e., application-level). The mechanism is the same for upper queue-level.

Let $\Lambda = \{\Lambda_1, \Lambda_2, \Lambda_3, \dots\}$ denote the set of submitted applications, and $\tilde{\Lambda}$ be the set of its active applications (the 'active' means there are pending or running tasks available). Let a_i be the arriving time for the application Λ_i . According to the *Early Bird Privilege* and

max-min fairness policy, the *PAR* $\phi_i(t)$ for the active application Λ_i should be,

$$\phi_i(t) = \begin{cases} \max_{\Lambda_k \in \tilde{\Lambda}} \{f_k(t) | a_k < a_i\} = \\ \max_{\Lambda_k \in \tilde{\Lambda}} \left\{ \int_0^t g'_k(t) dt + \phi_k(t) | a_k < a_i \right\}, & (a_i > \min_{\Lambda_k \in \tilde{\Lambda}} \{a_k\}). \\ 0, & \text{others.} \end{cases} \quad (5.10)$$

Let $n_i^p(t)$ denote the number of pending (i.e., runnable) tasks for the application Λ_i at time t . Let ω_i be the shared weight for the i^{th} application. Based on the weighted max-min fairness strategy and Formula (5.6), (5.9), (5.10), the application Λ_i to be chosen at time t for fair resource allocation should satisfy the following condition,

$$\frac{f_i(t)}{\omega_i} = \min_{\Lambda_k \in \tilde{\Lambda}} \left\{ \frac{f_k(t)}{\omega_k} | n_i^p(t) > 0 \right\}. \quad (5.11)$$

We name this fairness model **Fully Long-Term Max-Min Fairness Model (F-LTMM)**, as illustrated in Figure 5.1(a), considering that it is *memorizing* all the way since YARN system starts working.

However, in practice, sometimes we may not want the system to be fully long-term. Instead, we may want it to be long-term for a period of time (e.g., 24 hours). It motives us further to propose a time window-based long-term fairness model below.

Semi-Long-Term Max-Min Fairness Model (S-LTMM). The key idea is that, instead of fully memorizing resources all the time since the system starts working, we can divide system working time into a set of time windows (by default, we call the *time window* as *round*). Within the round (i.e., *Intra-Round Phase*), we adopt the fully long-term fairness model. When the system moves to the next round (i.e., *Inter-Round Phase*), it ignores all jobs' history information from the previous round and starts memorizing from the beginning. Characteristically speaking, it is a hybrid of fully long-term fairness model at intra-round phase and memoryless fairness model at inter-round phase.

Figure 5.1(b) illustrates the model. Let L denote the time length of a computation round, and t^s be the start time of the current computation round. Then we can compute t^s with the following formula,

$$t^s = \begin{cases} t^s + \lfloor \frac{t-t^s}{L} \rfloor \cdot L, & (t > 0). \\ 0, & (t = 0). \end{cases} \quad (5.12)$$

Moreover, all of the *F-LTMM*-related elements, including *Wall Clock*, *PAR* and accumulated consumed resources for each application, should be updated and counted from t^s instead. Then Formula (5.6) should be updated to be,

$$f_i(t) = \int_{t^s}^t g'_i(t) dt + \phi_i(t). \quad (5.13)$$

Unlike *F-LTMM* whose *Wall Clock* is just equal to the application's arriving time, the *Wall Clock* in *S-LTMM* is round-based, referring to a non-active period of an application since t^s , e.g., Λ_2 in Figure 5.1(b). To keep compatible, we propose a concept of *Round Arriving Time* $\check{a}_i$ for Λ_i , defined as the starting time point at which the application becomes active since t^s , e.g., t_5 for Λ_2 at Round 2 in Figure 5.1(b). It can be computed based on the following formula,

$$\check{a}_i = \begin{cases} a_i, & (t^s \leq a_i). \\ t^s, & (\exists j \in \tau_i(t), t_{i,j}^s \leq t^s < t_{i,j}^c). \\ \min_{j \in \tau_i(t)} \{t_{i,j}^s | t_{i,j}^s > t^s\}, & \text{others.} \end{cases} \quad (5.14)$$

Let $t_{i,j}^s, t_{i,j}^c$ denote the start time and finished time for the j^{th} task of the application Λ_i , respectively. Particularly, for the finished tasks of each application in *S-LTMM*, only the j^{th} task satisfying $t_{i,j}^c > t^s$ will count. According to the time quantum-based approach, we then have,

$$\alpha_{i,j}(t) = \begin{cases} t_{i,j}^c - \max\{t^s, t_{i,j}^s\}, & (t^s < t_{i,j}^c \leq t). \\ \max\{Q, t - \max\{t^s, t_{i,j}^s\}\}, & (t < t_{i,j}^c \leq t^s + L). \\ 0, & \text{others.} \end{cases} \quad (5.15)$$

where Q denotes the time quantum. And accordingly, Formula (5.10) should be updated to

$$\phi_i(t) = \begin{cases} \max_{\Lambda_k \in \tilde{\Lambda}} \{ \int_{t^s}^t g'_k(t) dt + \phi_k(t) | \check{a}_k < \check{a}_i \}, & (\check{a}_i > \min_{\Lambda_k \in \tilde{\Lambda}} \{\check{a}_k\}). \\ 0, & \text{others.} \end{cases} \quad (5.16)$$

Finally, by combining Formula (5.12), (5.15), (5.9), (5.16), (5.13), similar to *F-LTMM*, we can obtain *S-LTMM* by allocating resources to the application Λ_i subject to Formula (5.9) stringently at time t .

5.5.2 Design and Implementation of LTYARN

In YARN, the resources are organized into multiple queues with hierarchical tree structure. Each queue can represent an organization and the resources are shared among them. As illustrated in Figure 5.3, it shows an example of three-level structure (many more levels can be supported according to user's requirements). There is a root node called *Root Queue*. It distributes the resources of the whole system to the intermediate nodes called *Parent Queues*. Each parent queue further re-distributes resources into its sub-queues (parent queues or leaf queues) recursively until to the bottom nodes called *Leaf Queues*. Finally, users' submitted applications within the same leaf queue share the resources.

Figure 5.2: Overview of LTYARN.

Figure 5.2 shows an overview of the design and implementation of LTYARN. It consists of three key components: *Quantum Updater (QU)*, *Resource Controller (RC)*, and *Resource Allocator (RA)*. QU is responsible for updating the time quantum for each queue dynamically, based on running and completed tasks. RC manages the allocated resources for each application/queue and computes the accumulated resources periodically. RA performs the resource allocation based on the accumulated resources of each application/queue.

5.5.2.1 Quantum Updater (QU)

For LTYARN, the suitable value of Q is very important for *fairness convergency*, which refers to the convergency of unfair applications for their long-term resources at a time point and after that they fairly share the resources with each other. To achieve fast convergency, we need to make Q be close to the real execution time of tasks.

However, there are several challenges for getting the suitable value of Q in that, 1). the task execution time for an application is generally unknown in advance. One solution is to give an empirical value for Q based on the previously finished jobs' information. 2). However, different applications often have different task execution time. It implies that even if a single Q can be suitable for an application, it meanwhile can be an unsuitable value (e.g., too small or too large) for other applications. That is, a single static value of Q cannot meet the suitability requirements of multiple jobs (applications) at the same time. 3). Due to the possibility of varied types of applications in different queues for YARN in practice, ensuring that each queue owns a suitable Q for its own applications is necessary so that they do not interfere with each other.

Herein, we propose an adaptive task quantum policy, implemented in Quantum Updater. It is a multi-level self-tuning approach based on the customized structure of YARN's resource organization, as shown in Figure 5.3. The top-to-down data flow is a quantum value assignment process. It works when a new element (e.g., queue or application) is added. In contrast, the bottom-up data flow is a self-tuning procedure, which refreshes periodically by a fixed time interval (e.g, 1 second).

Initially, the system administrator provides a threshold value for root-level quantum Q_0 . When a new application is submitted to the system, it starts to perform up-to-down initialization process. First, it will check whether its parent queue is new one or not (Arrow (1) in Figure 5.3). If yes, it assigns its parent-queue quantum with root-queue quantum, e.g., $Q_{1,1} \leftarrow Q_0$. Next, it checks its sub-queues (e.g., leaf-queue) (Arrow (2) in Figure 5.3). If it is a new one, it assigns its sub-queue quantum with its parent-queue quantum, e.g., $Q_{2,1} \leftarrow Q_{1,1}$. Lastly, it initializes its application quantum with its leaf-queue quantum, e.g., $Q_{3,1} \leftarrow Q_{2,1}$ (Arrow (3) in Figure 5.3).

In contrast, Quantum Updater checks the system periodically for new finished tasks. When there is a map/reduce task finished, the bottom-to-up self-adjustment process begins to work automatically. First, it will update its application quantum with its average task completion time (Arrow (4) in Figure 5.3). Next, it updates its leaf-queue quantum with its average application quantum (Arrow (5) in Figure 5.3). Similarly, it updates its parent-queue quantum using the average value of its leaf-queue quantum (Arrow (6) in Figure 5.3).

Finally, the root-queue quantum is updated with the average value of parent-queue quantum (Arrow (7) in Figure 5.3).

Figure 5.3: The adaptive task quantum policy for YARN. The *up-to-down* data flow is a task time quantum initialization process for new applications. The *bottom-to-up* data flow is a quantum self-adjustment process for existing applications/queues.

5.5.2.2 Resource Controller (RC)

Resource Controller (RC) is the main component of LTYARN. Its principle responsibility is to manage and update the accumulated resources for each queue, needed by RA, on the basis of the model S-LTMM. It tracks the allocated resource (e.g., container in YARN) and the execution time for each task. Based on this information, it performs the resource updating work periodically (e.g., 1 second). Each time, it first updates the starting time of the current round based on Formula (5.12) and the round arriving time for each application based on Formula (5.14). Next based on time quantum-based approach, it estimates the assumed execution time for each running/completed task with the updated quantum value from QU, according to Formula (5.15). The current allocated resource for each task can then be estimated with Formula (5.7). After that, it estimates the Pseudo Accumulated Resources (PAR) for each application based on Formula (5.16). Finally, it updates the accumulated resource for each application/queue based on Formula (5.13).

5.5.2.3 Resource Allocator (RA)

Resource Allocator (RA) locates at each queue of different levels, as shown in Figure 5.3. It is triggered whenever there are pending tasks and idle resources. RA can now support FIFO, memoryless max-min fairness and long-term max-min fairness for each queue. Users can choose either of them accordingly. For long-term max-min fairness, it performs fair resource allocation for each application/queue with the provided resource information from RC, based on Formula (5.11). We provide two important configuration arguments for each queue, e.g., time quantum Q and round length L in the default configuration file, to meet different requirements for different queues. Moreover, we also support minimum (maximum) resource share for queues under long-term max-min fairness.

In practice, it is better for its root queue to use the long-term max-min fairness, considering that each of its sub-queues could be a user or an organization. We need to guarantee the resource-as-you-pay fairness (i.e., SLA) for them. For each parent-queue representing an organization, we should also adopt the long-term max-min fairness if its subqueues (i.e., members of the organization) require resource-as-you-pay fairness. In contrast, when a queue belongs to a user, there might be no need to ensure resource-as-you-pay fairness for its sub-queues. In that case, we can choose either memoryless max-min fairness, long-term max-min fairness or FIFO freely.

5.6 Experimental Evaluation

Hadoop Cluster: We have implemented LTYARN in the latest version of YARN-2.2.0. We deploy the YARN framework in a cluster consisting of 10 compute nodes, each with two Intel X5675 CPUs (6 CPU cores per CPU with 3.07 GHz), 24 GB DDR3 memory and 56 GB hard disks. We configure one node as master, and the other 9 nodes as slaves. For each slave node, we configure its resource capacity with 24 GB memory.

Macro-Benchmark: To evaluate our long-term fair scheduler LTYARN for YARN, we ran a macro-benchmark consisting of four different workloads:

- A MapReduce instance with a mix of small and large jobs based on the workload at the Facebook.

- A MapReduce instance running a set of large-sized batch jobs generated with Purdue MapReduce Benchmarks Suite[39].
- Hive running a series of TPC-H queries.
- Spark[101] running a series of machine learning applications.

Bin	Job Type	# Maps	# Reduces	# Jobs
1	rankings selection	1	NA	38
2	grep search	2	NA	18
3	uservisits aggregation	10	2	14
4	rankings selection	50	NA	10
5	uservisits aggregation	100	10	6
6	rankings selection	200	NA	6
7	grep search	400	NA	4
8	rankings-uservisits join	400	30	2
9	grep search	800	60	2

Table 5.3: Job types and sizes for each bin in our synthetic Facebook workloads.

Synthetic Facebook Workload. We synthesize Facebook workload based on the distribution of jobs sizes and inter-arrival time at Facebook in Oct. 2009 provided by Zaharia et. al. [100]. The workload consists of 100 jobs. We categorize them into 9 bins of job types and sizes, as listed in Table 5.3. It is a mix of large number of small-sized jobs (1 ~ 15 tasks) and small number of large-sized jobs (e.g., 800 tasks³). The job submission time is derived from one of SWIM’s Facebook workload traces (e.g., FB-2009_samples_24_times_1hr_1.tsv)[3]. The jobs are from Hive benchmark[6], containing four types of applications, i.e., rankings selection, grep search (selection), uservisits aggregation and rankings-uservisits join.

Purdue Workload. We select five benchmarks (e.g., WordCount, TeraSort, Grep, InvertedIndex, HistogramMovices) randomly from Purdue MapReduce Benchmarks Suite[39]. We use 40G wikipedia data[12] for WordCount, InvertedIndex and Grep, 40G generated data for TeraSort and HistogramMovices with their provided tools. To emulate a series of regular job submissions in a data warehouse, we submit these five jobs sequentially at a fixed interval of 3 mins to the system.

³We reduce the size of the largest jobs in [100] to have the workload fit our cluster size.

Hive / TPC-H. To emulate continuous analytic query, such as analysis of users' behavior logs, we ran TPC-H benchmark queries on hive[14]. 40GB data are generated with provided data tools. Four representative queries Q1, Q9, Q12, Q17 are chosen, each of which we create five instances. We write a script to launch each one after the previous one finished in a round robin fashion.

Spark. Latest version of Spark has supported its job to run on the YARN system. We consider two CPU-intensive machine learning algorithms, namely, kmeans and alternating least squares (ALS) with provided example benchmarks. We ran 10 instances of each algorithm, which are launched by a script that waits 2 minutes after each job completed to submit the next.

5.6.1 LTRF Resource Allocation Flow

Figure 5.4: Overview of detailed fairness resource allocation flow for LTRF.

To understand the dynamic history-based resource allocation mechanism of LTRF under LTYARN, we sample the resource demands, current allocated resources and accumulated

resources for four workloads over a short period of $0 \sim 260$ seconds, as illustrated in Figure 5.4. Figure 5.4(a) and 5.4(b) show the normalized results of the current resource demand and current allocated resources for each workload with respect to its current share. Figure 5.4(c) presents the normalized accumulated resources for four workloads with respect to the system capacity.

Figure 5.4(a) shows that different workloads have different resource demands over time. At beginning, Purdue, Spark and Hive / TPC-H have an overloaded demand period (e.g., Purdue: $24 - 131$, Spark: $28 - 118$, HIVE / TPC-H: $28 - 146$). Figure 5.4(b) shows the allocation details for each workload over time. We can see that, during the common overloaded period of $28 - 118$, the curves for Purdue, Spark and Hive / TPC-H are fluctuated, indicating that LTRF is dynamically adjusting the amount of resource allocation to each workload, instead of simply assigning each workload the same amount of resources like MLRF. Through dynamic adjusting, the accumulated resources for the three workloads are balanced (i.e., the curve is close to each other) during the period $80 - 118$, as shown in Figure 5.4(c). However, for Facebook workload, its overloaded period occurs from $204 - 260$. During this period, the Purdue workload is also overloaded, as shown in Figure 5.4(a). To achieve the accumulated resource fairness, LTRF allocated a large amount of resource to it (e.g., $3.85/4.0 = 96.25\%$ at point 222) shown in Figure 5.4(b), to make it catch up with others. We can see the accumulated resource results in Figure 5.4(c) that, during $204 - 260$, there is a significant increment for Facebook workload, whereas other workloads increase slightly.

5.6.2 Fairness Results

In Section 5.2.2, we have shown that a good sharing policy should be able to satisfy two key points: 1). can minimize the sharing loss (i.e., SLA guarantee) to make it close or equal to zero; 2). can maximize the sharing benefit as much as possible (i.e., Sharing incentive). This section makes a comparison between MLRF and LTRF on these two points.

We show the compared fairness results between MLRF and LTRF for four workloads over time in Figure 5.5. All results are relative to the static partition case (without sharing) with fairness degree of one and sharing benefit/loss degrees of zero. Figure 5.5(a) and

5.5(c) present the sharing benefit/loss degrees based on Formula (4) and (5), respectively, for MLRF and LTRF. Figure 5.5(b) and 5.5(d) show the detailed fairness degree for each queue (workload) over time. We have the following observations:

First, the sharing policies of both MLRF and LTRF can bring sharing benefits for queues (workloads). For example, both Facebook and Purdue workloads, illustrated in Figure 5.5(b) and 5.5(d) obtain benefits under the shared scenario. This is due to the sharing incentive property, i.e., each queue has an opportunity to consume more resources than her share at a time, better off running at most all of her shared partition in a non-shared partition system.

Second, it can also have the sharing loss problem easily for some queues if without a good sharing policy, i.e., whose total resources are worse than that in the non-shared case. In the shared computing system with SLA requirement, this is a very serious problem and thereby we should minimize the sharing loss as much as possible. The graph in Figure 5.5(a) and (c) show that, LTRF has a much better result than MLRF. Specifically, Figure 5.5(a) indicates that the sharing loss problem for MLRF is constantly available until all the workloads complete (e.g., ≈ -0.5 on average), contributed primarily by Spark and TPC-H workloads given by Figure 5.5(b). In contrast, there is no more sharing loss problem after 650 seconds for LTRF, i.e., all workloads get sharing benefits after that. The explanation is that, for MLRF, it does not consider historical resource allocation. Due to the varied demands for each workload over time, it easily occurs two extreme cases: 1). some workloads get much more resources over time (e.g., Facebook and Purdue workloads in Figure 5.5(b)); 2). some workloads obtain much less resources that without sharing over time (e.g., Spark and TPC-H workloads in Figure 5.5(b)). In contrast, LTRF is a history-based fairness resource allocation policy. It can dynamically adjust the allocation of resources to each queue in terms of their historical consumption so that each queue can obtain much close amount of total resources over time. Its *lending agreement* can avoid two extreme cases existing in MLRF.

Finally, regarding the sharing loss problem at the early stage (e.g., $0 \sim 650$ seconds) of LTRF in Figure 5.5(c), it is mainly due to the unavoidable waiting allocation problem at starting stage: a first coming and running workload possess all resources and leads late

(a) Sharing benefit/loss degree with MLRF based on Formula (4) and (5).

(b) Detailed fairness degree for four queues with MLRF based on Formula (3).

(c) Sharing benefit/loss degree with LTRF based on Formula (4) and (5).

(d) Detailed fairness degree for four queues with LTRF based on Formula (3).

Figure 5.5: Comparison of fairness results over time for each workload under MLRF and LTRF in YARN. All results are relative to the static partition scenario (without sharing) whose fairness degree is always one and sharing benefit/loss is zero. (a) and (c) show the overall benefit/loss relative to the non-sharing scenario. (b) and (d) present the detailed fairness degree for each queue: 1). A queue gets sharing benefit when its fairness degree is larger than one; 2). Otherwise, it arises sharing loss problem when a queue's fairness degree is below one.

arriving workloads need to wait for a while until some tasks complete and release resources (e.g., it is obvious in 0 – 226 seconds in Figure 5.5(c)). The problem does also exist in MLRF. But LTRF can smooth this problem until it disappear over time via *lending agreement*, while MLRF cannot.

5.6.3 Performance Results

Figure 5.6: The compared performance results for Static Partitioning, MLRF and LTRF.

Figure 5.6 presents the performance results (i.e., speedup) for four workloads under Static Partitioning, MLRF and LTRF, respectively. All results are normalized with respect to Static Partitioning. We see that, 1). the shared cases (i.e., MLRF and LTRF) can possibly achieve better performance than the non-shared case (i.e., Static Partitioning), or at least as fast in performance in the shared system as they do in their static partitioning system. For example, for Facebook and Purdue workloads, both MLRF and LTRF have much better performance results (e.g., 14% ~ 19% improvement for MLRF, and 10% ~ 23% for LTRF) than exclusively using a static partitioning system. The finding is consistent with previous works such as Mesos[57]. The performance gain is mainly due to the resource preemption of unneeded resources from other queues in a shared system. The statement can be validated by reviewing our fairness results in Figure 5.5(b) and 5.5(d) in Section 5.6.2. We note that, the fairness degrees for both Facebook and Purdue workloads are above one (i.e., get sharing

benefit) during the most of time, except at the beginning stage. For Spark and Hive / TPC-H workloads, MLRF and LTRF can be at least as fast as static partitioning. 2). There is no conclusive result regarding which one is absolutely better than the other between MLRF and LTRF. For example, MLRF is better than LTRF for Facebook by about 7% and Spark by about 2%. However, LTRF outperforms MLRF for Purdue workload by about 8% and TPC-H by about 10%.

5.6.4 Adaptive Task Quantum Policy Evaluation

Figure 5.7: The adaptive task quantum results for LTRF in one hour.

To demonstrate the importance and effectiveness of adaptive task quantum policy for YARN, this section performs a contrast experiment by showing the effects of accumulated resource results over time under the fixed time quantum and the adaptive task quantum mechanism proposed in Section 5.5.2.1.

We consider a scenario where the configured task quantum (e.g., 600s) is much larger than the real task execution time of workloads. Figure 5.7 shows the compared accumulated results for LTRF over time within one hour, which are normalized with respect to the system capacity. We have the following observations:

First, Figure 5.7(a) illustrates that the accumulated resource under the fixed task time quantum policy fluctuates up and down (i.e., unstable) significantly over time. Thus, it is not a good indicator for resource-as-you-pay fairness and fail to converge fast for fairness. This is due to the following assumption made about execution time in the quantum-based approach: 1). the assumed execution time for the completed task is equal to its real execution time; 2). for the running task, we compute its assumed execution time using the maximum value of the configured time quantum and its real execution time. Take Facebook workload for example. Its average task execution time is about 11s. At time 1439s, there are 107 running tasks, whose assumed execution time is 600, and its normalized accumulated resource is 1019. However, at time 1450s (i.e., after 11s), there are 31 running tasks, indicating that at least 76 tasks completed during this period, causing a significant drop for its normalized accumulated resource (e.g., 630).

In contrast, with adaptive task quantum policy, as shown in Figure 5.7(b), the curves of accumulated resource become much smoother, making it good as an indicator for resource-as-you-pay fairness. Figure 5.7(c) shows the adaptive task quantum results over time for four workloads. We see that each workload has varied task quantum and our policy can adjust them dynamically, validating the effectiveness of our adaptive approach.

5.7 Summary

Shared pay-as-you-go computing system has been becoming a trend across multiple users due to the benefit of resource sharing (i.e., sharing incentive property). However, this chapter finds that, the classical *memoryless* resource fairness policies, widely used in many existing popular frameworks and schedulers, including Hadoop, YARN, Mesos, Choosy, Quincy, DRF, are *not* suitable in pay-as-you-go computing system due to three serious problems, i.e., cost-inefficient workload submission problem, untruthfulness problem and

resource-as-you-pay unfairness problem. To address these problems, we propose LTRF and demonstrate that it is *suitable* for pay-as-you-go computing system. Besides, we also propose five payment-oriented properties as metrics to measure the quality for any fair policy in a pay-as-you-go computing system. As a case study, we put focus on max-min fairness and YARN specifically. We have developed *LTYARN*, a long-term YARN fair scheduler for the YARN (i.e., Hadoop-2.2.0) and our experiments demonstrate the effectiveness of our methods. Finally, we have made LTYARN open-source at <http://sourceforge.net/projects/ltyarn/>.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Hadoop is a popular batch processing system that supports MapReduce workloads (i.e., a batch of jobs) submitted from multiple users. The *performance* and *fairness* are two key metrics. In this thesis, we focus on optimization techniques of job scheduling and resource allocations for these two metrics.

The major contributions for this thesis are summarized as follows:

- *Job Ordering Optimization for MapReduce Workloads.* In Hadoop MRv1, due to the facts for MapReduce jobs that, 1) map tasks can only use map slots and reduce tasks can only use reduce slots, and 2) the general execution constraint that map tasks are executed before reduce tasks, different job submission orders can have significantly different performance under the FIFO scheduler. Therefore, we try to improve the performance of MapReduce workloads by optimizing their job submission orders. The benefit for this technique is that, it does not need to modify the Hadoop source code, which is useful for users to optimize their workloads in the case that they have no permission to change or upgrade the Hadoop system. We focus on two performance metrics, namely, *makespan* (i.e., the sum of completion time of all jobs) and *total completion time* (i.e., the sum of completion time of all jobs), which are both important and meaningful in practice. Generally, the makespan is often used to measure the performance and utilization efficiency of a system, which only considers the computation time of jobs. In contrast, the total completion time is a generalized makespan

with queuing time (i.e., waiting time) included, which is often used 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).

First, we have proposed two job ordering optimization algorithms for offline MapReduce workloads whose jobs arrive at the same time. One algorithm is to optimize the makespan for a MapReduce workload. Another is a bi-criteria optimization algorithm for both makespan and total completion time. Moreover, a theoretical studies on the deduction of makespan and optimized job orders are also given.

Second, we have considered the online MapReduce workloads whose jobs can arrive over time. A flexible job ordering optimization system called *MROrder* is proposed. It contains two key components, namely, *policy module* and *ordering engine*. The policy module decides when and how to perform job ordering for online MapReduce jobs. The ordering engine performs the job ordering work. By adding different heuristic offline job ordering algorithms into ordering engine, we can perform the job ordering optimization for various metrics, e.g., makespan, total completion time.

We have performed numerical experiments to evaluate the proposed algorithms and system. The results show that our job ordering optimization approaches can significantly improve the performance for offline and online MapReduce workloads.

- *Dynamic Slot Optimization for MapReduce Clusters*. We have proposed an optimized slot allocation system called DynamicMR to fundamentally improve the performance of MapReduce jobs without needing any information or assumption. It is achieved through modifying the Hadoop source code. It consists of three different optimization techniques. The first technique is called *Dynamic Hadoop slot allocation (DHSA)*. It improves the slot utilization by breaking the slot allocation constraint to allow map tasks can run on reduce slots and vice versa. The second technique is *Speculative Execution Performance Balancing (SEPB)*. It observes that speculative execution can improve the performance of a single job, but at the expense of cluster efficiency for the whole jobs. It is proposed to balance the performance tradeoff between a single

jobs execution time and a batch of jobs execution time by determining dynamically when it is time to schedule and allocate slots for speculative tasks. The third one is called *Slot PreScheduling*. It improves the performance by maximizing the data locality. In contrast to delay scheduling that improves the data locality at the expense of fairness, *Slot PreScheduling* that can improve the data locality but has no negative impact on fairness. It is achieved at the expense of load balance between slave nodes. By incorporating these techniques together, our DynamicMR can optimize the performance for MapReduce workloads comprehensively. The benefits of DynamicMR include: 1). it can suite any schedulers, such as FIFO, fair, capacity schedulers; 2). it can work for any types of applications, e.g., independent or dependent jobs. 3). it have a set of complete optimization approaches towards a MapReduce workload in comparison to a ad-hoc one.

We have made numerical studies to evaluate DynamicMR. The experimental results show that our DynamicMR can improve the performance of Hadoop MRv1 significantly while maintaining the fairness. Moreover, we make a comparison with YARN experimentally. The results show that DynamicMR outperforms YARN consistently for multiple jobs.

- *Long-Term YARN Fair Scheduling for Pay-as-you-use Computing Systems.* It has been becoming a trend for *shared* pay-as-you-use computing across multiple clients due to the improved resource utilization and cost efficiency of resource sharing (i.e., sharing incentive property). However, we note that, the fair policies adopted by existing frameworks such as YARN, Dryad, are all memoryless, i.e., current allocation does not depend on history information, which are not suitable for pay-as-you computing system. Particularly, we show three severe problems including trivial workload submission problem, strategy-proofness problem, and resources-as-you-pay fairness problem. To address it, we propose long-term resource fairness (LTRF) and implement it in YARN by developing a scheduler LTYARN. We show that LTRF can meet several desirable properties needed by any payment-oriented computing system.

We have performed lots of experiments to evaluate our approach. The results show that it leads to a better resource fairness than existing memoryless fair scheduler.

6.2 Future Works

In this section, we present some optimization work that we plan to do in future as follows:

- *Efficient Resource Management for YARN (i.e., Hadoop MRv2)*. In this thesis, we have considered the performance optimization for Hadoop MRv1 in Chapter 3 and Chapter 4. Due to the pre-configuration of map slots and reduce slots and strict execution constrain that map tasks run before reduce tasks per slave node, the slot resources can be poorly utilized for the slot-based Hadoop MRv1. In view of this, YARN drops the slot-based model and proposes a new concept of 'container' consisting of a package of resources, where either map tasks or reduce tasks can run on. The resource utilization can be thereby improved significantly. However, we observe that, based on the current container allocation model, there are still some factors affecting the resource utilization and performance.
 - (I). *User's Resource Pre-configuration of Container*. In YARN, for each application (e.g., MapReduce job), the user needs to have a configuration for the container regarding each type of resource that each task will need. Then the system allocates resources based on it. However, in practice, the consumed resources at runtime are prone to be either smaller or much larger than the configured one due to improper configuration. When the consumed resource is larger than requested one, it will kill the container/task automatically, wasting computation. In contrast, when the consumed resource is smaller than the requested one, the idle parts will be wasted. Particularly, the task will be killed automatically if there a certain type of resources configured insufficiently under the multi-resource configuration (e.g., memory, CPU). Besides, improper configuration will also result in dominant resource problem.

(II). *Varied Resource Demands Over Time for Tasks*. Figure 6.1 shows the memory consumption for a YARN task over time. We see that, the resource consumption for a task is not static, instead, it changes over time. Thus, it is difficult for users to have a proper configuration for containers.

Figure 6.1: The demand of memory resource for a task is changing over time.

Therefore, we plan to propose some optimization techniques to efficiently improve resource utilization for YARN by taking into account these factors.

- *Multi-resource Fairness Allocation on Pay-as-you-go Computing Systems*. In chapter 5, we have proposed a long-term resource fairness and implemented it in YARN by developing a LTYARN scheduler for pay-as-you-go computing system. It currently only considers the single resource fairness, e.g., memory. However, in practice, the resource demands of tasks are often *heterogeneous*. In Figure 6.2, it shows a resource usage profile of tasks from Google in a data center of 12 thousands of machines based on the Google trace[4] over about a month-long period (May 2011). The position of a circle indicates the CPU and memory resources consumed by tasks. The size of a circle is logarithmic to the number of tasks in the region of the circle. It shows that there are significantly varied demands for tasks on CPU and memory resources. Thus, to efficiently allocate resources, we need to consider *multi-resource allocation* that takes multiple resource types (e.g., cpu, memory, i/o) into account[42].

Dominant Resource Fairness (DRF)[42] is one of the most popular multi-resource allocation policies for cluster environments. It has been widely used in existing sys-

Figure 6.2: Heterogeneous resource demands for tasks from google traces[4].

tems such as YARN[91] and Mesos[57]. It introduces the concept of a user's *dominant share*, which is the highest share of any typed resource that the user has been allocated. The resource corresponding to the dominant share is called *dominant resource*. The multi-resource fairness is achieved by equalizing the dominant shares across all users at the point of time considering resource allocation (i.e., *memoryless*). Unfortunately, due to its '*memoryless*' allocation feature, it fails to meet some of the desired properties in Section 5.2.1 of Chapter 5. It indicates that DRF is *not* suitable for shared pay-as-you-go computing system.

To propose a multi-resource fair allocation for pay-as-you-go computing system, one solution is to extend DRF with long-term resource fairness policy. Thereby, it remains our future work to extend our current work of long-term resource fairness for multiple resource types.

Author's Publications

Papers Published

- (i) **Shanjiang Tang**, Bu-Sung Lee, Bingsheng He, Dynamic Job Ordering and Slot Configurations for MapReduce Workloads, *IEEE Transactions on Services Computing*, 2015.
- (ii) **Shanjiang Tang**, Bu-Sung Lee, and Bingsheng He, Towards Economic Fairness for Big Data Processing in Pay-as-You-Go Cloud Computing. *In 2014 IEEE 6th International Conference on Cloud Computing Technology and Science (CloudCom'14)*, pp. 638-643, Singapore, Dec. 2014.
- (iii) **Shanjiang Tang**, Bu-Sung Lee, and Bingsheng He, DynamicMR A Dynamic Slot Allocation Optimization Framework for MapReduce Clusters, *IEEE Transaction on Cloud Computing*, vol. 2, no.3, pp.333-347, July. 2014.
- (iv) **Shanjiang Tang**, Bu-Sung Lee, Bingsheng He and Haikun Liu, Long-Term Resource Fairness: Towards Economic Fairness on Pay-as-you-use Computing Systems. *In Proceedings of the 28th ACM International Conference on Supercomputing, ICS 14*, pp. 251260, Munich, Germany, June. 2014.
- (v) **Shanjiang Tang**, Bu-Sung Lee, Bingsheng He, Dynamic slot allocation technique for MapReduce clusters. *2013 IEEE International Conference on Cluster Computing*, pp. 1-8, Indianapolis, 2013.
- (vi) **Shanjiang Tang**, Bu-Sung Lee, Bingsheng He, MROrder: Flexible Job Ordering Optimization for Online MapReduce Workloads. *Euro-Par'13*, pp 291-304, Aachen, Germany, 2013.

- (vii) **Shanjiang Tang**, Ce Yu, Jizhou Sun, Bu-Sung Lee, Tao Zhang, Zheng Xu, and Huabei Wu, EasyPDP: An Efficient Parallel Dynamic Programming Runtime System for Computational Biology, *IEEE Transactions on Parallel and Distributed System*, vol. 23, no. 5, pp. 862-872, May. 2012.
- (viii) **Shanjiang Tang**, Bu-Sung Lee, Bingsheng He, Speedup for Multi-Level Parallel Computing. *IPDPS Workshops 2012*, pp. 537-546, Shanghai, May. 2012.
- (ix) **Shanjiang Tang**, Ce Yu, Bu-Sung Lee, Chao Sun, and Jizhou Sun, Adaptive Data Refinement for Parallel Dynamic Programming Applications, *IPDPS Workshops 2012*, pp. 2220-2229, Shanghai, May. 2012.

Papers under Review

- (i) **Shanjiang Tang**, Bu-Sung Lee, and Bingsheng He, Economic Fairness for Resource Sharing in Pay-as-you-use Cloud Computing, submitted to *IEEE Transactions on Knowledge and Data Engineering*.
- (ii) **Shanjiang Tang**, Zhaojie Niu, Bu-Sung Lee, and Bingsheng He, Multi-Resource Fair Allocation in Pay-as-you-go Cloud Computing, submitted to *SC 2015*.

References

- [1] Amazon ec2. In <http://aws.amazon.com/ec2>. 32, 49
- [2] Capacity scheduler guide. In http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html. 2
- [3] Facebook workload traces. In <https://github.com/SWIMProjectUCB/SWIM/wiki/Workloads-repository>. 111
- [4] Google cluster data. In <https://code.google.com/p/googleclusterdata/>. viii, 124, 125
- [5] Hadoop. In <http://hadoop.apache.org/>. 1, 94
- [6] Hive performance benchmarks. In <https://issues.apache.org/jira/browse/HIVE-396>. 111
- [7] Howmanymapsandreduces. In http://wiki.apache.org/hadoop/HowMany_MapsAndReduces. 2, 32
- [8] Loan agreement. In http://en.wikipedia.org/wiki/Loan_agreement. 96
- [9] Lognormal distribution. In http://en.wikipedia.org/wiki/Log-normal_distribution. 35
- [10] Max-min fairness (wikipedia). In http://en.wikipedia.org/wiki/Max-min_fairness. 5, 18, 56, 103
- [11] Poisson distribution. In http://en.wikipedia.org/wiki/Poisson_distribution. 45
- [12] Puma datasets. In <https://sites.google.com/site/farazahmad/pumadatasets>. 77, 111
- [13] The scheduling problem. In <http://riot.ieor.berkeley.edu/Applications/Scheduling/algorithms.html>. 29
- [14] Tpc-h benchmark on hive. In <https://issues.apache.org/jira/browse/HIVE-600>. 112

- [15] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *Proc. VLDB Endow.*, 1(1):958–969, Aug. 2008. 14
- [16] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB ’01, pages 169–180, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. 15
- [17] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris. Reining in the outliers in map-reduce clusters using mantri. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI’10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. 11
- [18] H. Arabnejad and J. Barbosa. Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems. In *Proceedings of the 2012 IEEE 10th International Symposium on Parallel and Distributed Processing with Applications*, ISPA ’12, pages 633–639, Washington, DC, USA, 2012. IEEE Computer Society. 17
- [19] S. Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC ’10, pages 137–142, New York, NY, USA, 2010. ACM. 16
- [20] J. Błażewicz, M. Machowiak, J. Węglarz, M. Kovalyov, and D. Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan. *Annals of Operations Research*, 129(1-4):65–80, 2004. 9
- [21] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC ’11, pages 7:1–7:14, New York, NY, USA, 2011. ACM. 13
- [22] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical scheduling for diverse datacenter workloads. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC ’13, pages 4:1–4:15, New York, NY, USA, 2013. ACM. 18, 19
- [23] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. The haloop approach to large-scale iterative data analysis. *The VLDB Journal*, 21(2):169–190, Apr. 2012. 15

- [24] Q. Chen, C. Liu, and Z. Xiao. Improving mapreduce performance using smart speculative execution strategy. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013. 11
- [25] S. Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *Proc. VLDB Endow.*, 3(1-2):1459–1468, Sept. 2010. 15
- [26] Y. Chen, A. Ganapathi, R. Griffith, and R. Katz. The case for evaluating mapreduce performance using workload suites. In *Proceedings of the 2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*, MASCOTS '11, pages 390–399, Washington, DC, USA, 2011. IEEE Computer Society. 44
- [27] W. Cirne and F. Berman. When the herd is smart: Aggregate behavior in the selection of job request. *IEEE Trans. Parallel Distrib. Syst.*, 14(2):181–192, Feb. 2003. 30
- [28] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 21–21, Berkeley, CA, USA, 2010. USENIX Association. 16
- [29] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, Jan. 2008. 1
- [30] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *Symposium Proceedings on Communications Architectures & Protocols*, SIGCOMM '89, pages 1–12, New York, NY, USA, 1989. ACM. 18
- [31] J. Dittrich, J.-A. Quiané-Ruiz, A. Jindal, Y. Kargin, V. Setty, and J. Schad. Hadoop++: Making a yellow elephant run like a cheetah (without it even noticing). *Proc. VLDB Endow.*, 3(1-2):515–529, Sept. 2010. 14
- [32] J. Dittrich, J.-A. Quiané-Ruiz, S. Richter, S. Schuh, A. Jindal, and J. Schad. Only aggressive elephants are fast elephants. *Proc. VLDB Endow.*, 5(11):1591–1602, July 2012. 14
- [33] C. Doukeridis and K. Norvag. A survey of large-scale analytical query processing in mapreduce. *The VLDB Journal*, pages 1–26, 2013. 9
- [34] M. Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009. 17, 18

- [35] P.-F. Dutot, L. Eyraud, G. Mounié, and D. Trystram. Bi-criteria algorithm for scheduling jobs on cluster platforms. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '04, pages 125–132, New York, NY, USA, 2004. ACM. 10
- [36] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *Proc. VLDB Endow.*, 5(6):586–597, Feb. 2012. 13
- [37] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson. Co-hadoop: Flexible data placement and its exploitation in hadoop. *Proc. VLDB Endow.*, 4(9):575–585, June 2011. 14
- [38] L. Fan, F. Zhang, G. Wang, and Z. Liu. An effective approximation algorithm for the malleable parallel task scheduling problem. *J. Parallel Distrib. Comput.*, 72(5):693–704, May 2012. 9
- [39] A. Faraz, L. Seyong, T. Mithuna, and V. T. N. Puma: Purdue mapreduce benchmarks suite. Technical Report EECS-2012-Oct, School of Electrical and Computer Engineering, Purdue University, Oct 2012. 77, 111
- [40] A. Floratou, J. M. Patel, E. J. Shekita, and S. Tata. Column-oriented storage techniques for mapreduce. *Proc. VLDB Endow.*, 4(7):419–429, Apr. 2011. 15
- [41] P. François Dutot, G. Mouni, and D. Trystram. Scheduling parallel tasks approximation algorithms, 2003. 9
- [42] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 24–24, Berkeley, CA, USA, 2011. USENIX Association. 18, 92, 94, 124
- [43] A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Choosy: Max-min fair sharing for datacenter jobs with constraints. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 365–378, New York, NY, USA, 2013. ACM. 18, 94
- [44] B. Gufler, N. Augsten, A. Reiser, and A. Kemper. Load balancing in mapreduce based on scalable cardinality estimates. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 522–533, April 2012. 12

- [45] Z. Guo, G. Fox, and M. Zhou. Investigation of data locality and fairness in mapreduce. In *Proceedings of Third International Workshop on MapReduce and Its Applications Date*, MapReduce '12, pages 25–32, New York, NY, USA, 2012. ACM. 11
- [46] Z. Guo, G. Fox, and M. Zhou. Investigation of data locality in mapreduce. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 419–426, Washington, DC, USA, 2012. IEEE Computer Society. 11
- [47] Z. Guo, G. Fox, M. Zhou, and Y. Ruan. Improving resource utilization in mapreduce. In *Proceedings of the 2012 IEEE International Conference on Cluster Computing*, CLUSTER '12, pages 402–410, Washington, DC, USA, 2012. IEEE Computer Society. 11
- [48] J. Gupta, A. Hariri, and C. Potts. Scheduling a two-stage hybrid flow shop with parallel machines at the first stage. *Annals of Operations Research*, 69(0):171–191, 1997. 10
- [49] J. N. D. Gupta. Two-stage, hybrid flowshop scheduling problem. *The Journal of the Operational Research Society*, 39(4):pp. 359–364, 1988. 10
- [50] J. N. D. Gupta, K. Hennig, and F. Werner. Local search heuristics for two-stage flow shop problems with secondary criterion. *Comput. Oper. Res.*, 29(2):123–149, Feb. 2002. 10
- [51] M. Hammoud, M. S. Rehman, and M. F. Sakr. Center-of-gravity reduce task scheduling to lower mapreduce network traffic. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 49–58, Washington, DC, USA, 2012. IEEE Computer Society. 11
- [52] M. Hammoud and M. F. Sakr. Locality-aware reduce task scheduling for mapreduce. In *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, CLOUDCOM '11, pages 570–576, Washington, DC, USA, 2011. IEEE Computer Society. 11
- [53] C. He, D. Weitzel, D. Swanson, and Y. Lu. Hog: Distributed hadoop mapreduce on the grid. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 1276–1283, Washington, DC, USA, 2012. IEEE Computer Society. 1

- [54] Y. He, R. Lee, Y. Huai, Z. Shao, N. Jain, X. Zhang, and Z. Xu. Rcfile: A fast and space-efficient data placement structure in mapreduce-based warehouse systems. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering, ICDE '11*, pages 1199–1208, Washington, DC, USA, 2011. IEEE Computer Society. 15
- [55] S. R. Hejazi*y and S. Saghafianz. Flowshop-scheduling problems with makespan criterion: a review, 2004. 10
- [56] 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 *In CIDR*, pages 261–272, 2011. 16
- [57] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation, NSDI'11*, pages 22–22, Berkeley, CA, USA, 2011. USENIX Association. 18, 94, 116, 125
- [58] O. ilavsk. Net present value approach: Method for economic assessment of innovation projects. *Procedia - Social and Behavioral Sciences*, 156(0):506 – 512, 2014. 19th International Scientific Conference Economics and Management 2014 (ICEM-2014). 98
- [59] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2Nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007, EuroSys '07*, pages 59–72, New York, NY, USA, 2007. ACM. 94
- [60] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09*, pages 261–276, New York, NY, USA, 2009. ACM. 18
- [61] M.-Y. Iu and W. Zwaenepoel. Hadooptosql: A mapreduce query optimizer. In *Proceedings of the 5th European Conference on Computer Systems, EuroSys '10*, pages 251–264, New York, NY, USA, 2010. ACM. 16
- [62] E. Jahani, M. J. Cafarella, and C. Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, Mar. 2011. 16

- [63] K. Jansen and H. Zhang. Scheduling malleable tasks with precedence constraints. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '05, pages 86–95, New York, NY, USA, 2005. ACM. 9
- [64] S. Johnson. Optimal two- and three-stage production schedules with setup times included. In *Naval Research Logistics*, SPAA '11, pages 61–C68. Wiley, 2006. 25, 26, 27, 43
- [65] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for mapreduce. In *Proceedings of the Twenty-first Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '10, pages 938–948, Philadelphia, PA, USA, 2010. Society for Industrial and Applied Mathematics. 10
- [66] I. Kash, A. D. Procaccia, and N. Shah. No agent left behind: Dynamic fair division of multiple resources. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, AAMAS '13, pages 351–358, Richland, SC, 2013. International Foundation for Autonomous Agents and Multiagent Systems. 18, 19
- [67] L. Kolb, A. Thor, and E. Rahm. Load balancing for mapreduce-based entity resolution. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, pages 618–629, Washington, DC, USA, 2012. IEEE Computer Society. 12
- [68] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skew-resistant parallel processing of feature-extracting scientific user-defined functions. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 75–86, New York, NY, USA, 2010. ACM. 13
- [69] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia. Skewtune: Mitigating skew in mapreduce applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 25–36, New York, NY, USA, 2012. ACM. 12, 13, 69
- [70] G. J. Kyparisis and C. Koulamas. A note on makespan minimization in two-stage flexible flow shops with uniform machines. *European Journal of Operational Research*, 175(2):1321 – 1327, 2006. 10

- [71] J. Leung, L. Kelly, and J. H. Anderson. *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*. CRC Press, Inc., Boca Raton, FL, USA, 2004. 9
- [72] Y. Lin, D. Agrawal, C. Chen, B. C. Ooi, and S. Wu. Llama: Leveraging columnar storage for scalable join processing in the mapreduce framework. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 961–972, New York, NY, USA, 2011. ACM. 15
- [73] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós. On scheduling in map-reduce and flow-shops. In *Proceedings of the Twenty-third Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '11, pages 289–298, New York, NY, USA, 2011. ACM. 10, 23
- [74] J. Ngubiri and M. van Vliet. A metric of fairness for parallel job schedulers. *Concurr. Comput. : Pract. Exper.*, 21(12):1525–1546, Aug. 2009. 17
- [75] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. Mrshare: Sharing across multiple queries in mapreduce. *Proc. VLDB Endow.*, 3(1-2):494–505, Sept. 2010. 13
- [76] C. O?uz and M. F. Ercan. Scheduling multiprocessor tasks in a two-stage flow-shop environment. *Computers & Industrial Engineering*, 33(1C2):269 – 272, 1997. Proceedings of the 21st International Conference on Computers and Industrial Engineering. 10
- [77] C. O?uz, M. F. Ercan, T. E. Cheng, and Y. Fung. Heuristic algorithms for multiprocessor task scheduling in a two-stage hybrid flow-shop. *European Journal of Operational Research*, 149(2):390 – 403, 2003. Sequencing and Scheduling. 10
- [78] B. Palanisamy, A. Singh, L. Liu, and B. Jain. Purlieus: Locality-aware resource allocation for mapreduce in a cloud. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 58:1–58:11, New York, NY, USA, 2011. ACM. 1, 11
- [79] D. C. Parkes, A. D. Procaccia, and N. Shah. Beyond dominant resource fairness: Extensions, limitations, and indivisibilities. In *Proceedings of the 13th ACM Conference on Electronic Commerce*, EC '12, pages 808–825, New York, NY, USA, 2012. ACM. 18, 19

- [80] C. Rajendran. Two-stage flowshop scheduling problem with bicriteria. *The Journal of the Operational Research Society*, 43(9):pp. 871–884, 1992. 10
- [81] S. R. Ramakrishnan, G. Swart, and A. Urmanov. Balancing reducer skew in mapreduce workloads using progressive sampling. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 16:1–16:14, New York, NY, USA, 2012. ACM. 12
- [82] S. Rao, R. Ramakrishnan, A. Silberstein, M. Ovsiannikov, and D. Reeves. Sailfish: A framework for large scale data processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM. 13
- [83] A. Rasmussen, V. T. Lam, M. Conley, G. Porter, R. Kapoor, and A. Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, New York, NY, USA, 2012. ACM. 13
- [84] G. Sabin and P. Sadayappan. Unfairness metrics for space-sharing parallel job schedulers. In *Proceedings of the 11th International Conference on Job Scheduling Strategies for Parallel Processing*, JSSPP'05, pages 238–256, Berlin, Heidelberg, 2005. Springer-Verlag. 17
- [85] P. Sanders and J. Speck. Efficient parallel scheduling of malleable tasks. In *Proceedings of the 2011 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '11, pages 1156–1166, Washington, DC, USA, 2011. IEEE Computer Society. 9
- [86] I. Shadi, J. Hai, L. Lu, and H. Bingsheng. Adaptive disk i/o scheduling for mapreduce in virtualized environment. In *2011 International Conference on Parallel Processing (ICPP)*, ICPP '11, pages 335 – 344. IEEE Computer Society, 2011. 44
- [87] J. Tan, S. Meng, X. Meng, and L. Zhang. Improving reducetask data locality for sequential mapreduce jobs. In *INFOCOM, 2013 Proceedings IEEE*, pages 1627–1635, April 2013. 11
- [88] J. Tan, X. Meng, and L. Zhang. Coupling task progress for mapreduce resource-aware scheduling. In *INFOCOM, 2013 Proceedings IEEE*, pages 1618–1626, April 2013. 11

- [89] S. Tang, B.-S. Lee, and B. He. Mrorder: Flexible job ordering optimization for online mapreduce workloads. In F. Wolf, B. Mohr, and D. Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 291–304. Springer Berlin Heidelberg, 2013. 10, 61
- [90] 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 *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 251–260, New York, NY, USA, 2014. ACM. 18, 19, 92
- [91] V. K. Vavilapalli, A. C. Murthy, C. Douglas, S. Agarwal, M. Konar, R. Evans, T. Graves, J. Lowe, H. Shah, S. Seth, B. Saha, C. Curino, O. O'Malley, S. Radia, B. Reed, and E. Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM. 18, 94, 125
- [92] A. Verma, L. Cherkasova, and R. H. Campbell. Play it again, simmr! In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 253–261, Washington, DC, USA, 2011. IEEE Computer Society. 35
- [93] 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 *Proceedings of the 2012 IEEE 20th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS '12, pages 11–18, Washington, DC, USA, 2012. IEEE Computer Society. 10, 24, 27
- [94] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the 1st USENIX Conference on Operating Systems Design and Implementation*, OSDI '94, Berkeley, CA, USA, 1994. USENIX Association. 18
- [95] W. Wang, B. Li, and B. Liang. Dominant resource fairness in cloud computing systems with heterogeneous servers. In *INFOCOM, 2014 Proceedings IEEE*, pages 583–591, April 2014. 18, 19, 92
- [96] X. Wang, C. Olston, A. D. Sarma, and R. Burns. Coscan: Cooperative scan sharing in the cloud. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, SOCC '11, pages 11:1–11:12, New York, NY, USA, 2011. ACM. 14

- [97] T. White. *Hadoop: The Definitive Guide*. O'Reilly, first edition edition, june 2009. 11, 12, 18, 64, 65, 66
- [98] J. Wolf, A. Balmin, D. Rajan, K. Hildrum, R. Khandekar, S. Parekh, K.-L. Wu, and R. Vernica. On the optimization of schedules for mapreduce workloads in the presence of shared scans. *The VLDB Journal*, 21(5):589–609, Oct. 2012. 14
- [99] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, EECS Department, University of California, Berkeley, Apr 2009. 34, 35, 55
- [100] 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 *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 265–278, New York, NY, USA, 2010. ACM. 2, 11, 35, 44, 45, 67, 111
- [101] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association. 111
- [102] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 29–42, Berkeley, CA, USA, 2008. USENIX Association. 11, 64, 65, 66, 83
- [103] H. Zhao and R. Sakellariou. Scheduling multiple dags onto heterogeneous systems. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing*, IPDPS'06, pages 159–159, Washington, DC, USA, 2006. IEEE Computer Society. 17

Appendix A

Theorem and Lemma in Chapter 3

In this part, we present the detailed proofs for theorems and lemmas listed in Chapter 3.

A.1 Proof of Lemma 3.2

Lemma 3.2 *Given a job order ϕ , there is a upper bound makespan denoted by $\hat{C}_{max}$ (i.e. $C_{max} \leq \hat{C}_{max}$) for the generalized case as follows:*

$$\hat{C}_{max} = \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k T_i^{\mathcal{M}} + \max_{1 \leq i \leq k} \{\hat{t}_i^{\mathcal{M}}\} + \sum_{i=k}^n T_i^{\mathcal{R}} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\} \right\}.$$

where $T_i^{\mathcal{M}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|S^{\mathcal{M}}|}$, $T_i^{\mathcal{R}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|S^{\mathcal{R}}|}$, $\hat{t}_i^{\mathcal{M}} = \max_{1 \leq j \leq |J_i^{\mathcal{M}}|} \hat{t}_{i,j}^{\mathcal{M}}$ and $\hat{t}_i^{\mathcal{R}} = \max_{1 \leq j \leq |J_i^{\mathcal{R}}|} \hat{t}_{i,j}^{\mathcal{R}}$.

Proof Lemma 3.1 gives us an important implication regarding the *makespan* estimation for the simplified case. That is, finding a job J_{k_0} such that $X_{k_0} > 0$ and $X_{k'} = 0$ (i.e., no idle period in the reduce phase after the reduce tasks of J_{k_0} starts) for all $k_0 < k' \leq n$. Then the *makespan* is equal to $\sum_{i=1}^{k_0} T_i^{\mathcal{M}} + \sum_{i=k_0}^n T_i^{\mathcal{R}}$. It is also applicable for the generalized case.

Let $c_i^{\mathcal{M}}$ be the completion time at the map phase and $c_i^{\mathcal{R}}$ be the completion time at the reduce phase for the generalized case. Then we have $c_i^{\mathcal{M}} < \sum_{j=1}^i \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|S^{\mathcal{M}}|} + \max_{1 \leq j \leq i} \{\hat{t}_j^{\mathcal{M}}\}$. For the generalized case, there are two types of idle periods in the reduce phase, i.e., *partial idle period (PIP)* (e.g. X_3 in Figure 3.3 for Job J_3) and *full idle period (FIP)* (e.g., X_4 in Figure 3.3 for Job J_4), unlike the simplified case that has *FIP* only. Moreover, it is worth noting that for the simplified case, the job execution order is the same between in the map phase and in reduce phase. Whereas it does not always hold for the generalized case. In Figure 3.3, for example, the reduce tasks of J_2 start earlier than that of J_1 although the map tasks of J_1 start earlier in the map phase. Therefore, we need to consider the following two cases:

(1). Suppose the execution order in the map phase is the same as in the reduce phase for the generalized case. Let's assume that Job J_{k_0} is the splitting point that $X_{k_0} > 0$ (either *FIP* or *FIP*) and $X_{k'} = 0$ (no idle period) for all the remaining jobs $J_{k'} (k_0 < k' \leq n)$. Then we have $C_{max} < c_{k_0}^M + \sum_{i=k_0}^n \frac{\sum_{j=1}^{|J_i^M|} t_{i,j}^M}{|S^R|} + \max_{1 \leq i \leq n} \{\hat{t}_i^R\} < \sum_{i=1}^{k_0} \frac{\sum_{j=1}^{|J_i^M|} t_{i,j}^M}{|S^M|} + \max_{1 \leq i \leq k_0} \{\hat{t}_i^M\} + \sum_{i=k_0}^n \frac{\sum_{j=1}^{|J_i^R|} t_{i,j}^R}{|S^R|} + \max_{1 \leq i \leq n} \{\hat{t}_i^R\} \leq \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k \frac{\sum_{j=1}^{|J_i^M|} t_{i,j}^M}{|S^M|} + \max_{1 \leq i \leq k} \{\hat{t}_i^M\} + \sum_{i=k}^n \frac{\sum_{j=1}^{|J_i^R|} t_{i,j}^R}{|S^R|} + \max_{1 \leq i \leq n} \{\hat{t}_i^R\} \right\}$.

(2). On the other hand, it is possible that there are out of order jobs in the reduce phase for the generalized case (e.g. job order in Figure 3.3). Note that the cause of the out of order at the reduce phase is due to the map tasks of later submitted job finish first. It can have the reduce tasks begin earlier in the reduce phase, making the maximum completion time for a batch of jobs less than that of the ordered case. In other words, if there is an out of order interval from the i^{th} job to the j^{th} job at the reduce phase, we can have $C_{max}^j \leq C'_{max}^j$, where C_{max}^j denotes the maximum completion time for j jobs in the out of order case and C'_{max}^j denotes the maximum completion time for j jobs in the ordered case. It may in turn cause later jobs to complete earlier (i.e. $C_{max}^k \leq C'_{max}^k$ for $k > j$). Hence, we have $C_{max} < \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k \frac{\sum_{j=1}^{|J_i^M|} t_{i,j}^M}{|S^M|} + \max_{1 \leq i \leq k} \{\hat{t}_i^M\} + \sum_{i=k}^n \frac{\sum_{j=1}^{|J_i^R|} t_{i,j}^R}{|S^R|} + \max_{1 \leq i \leq n} \{\hat{t}_i^R\} \right\}$ for this case, based on the case (1).

Finally, our proof completes based on (1) and (2). □

A.2 Proof of Lemma 3.3

Lemma 3.3 Let $\check{\phi}^*$ denote the job order produced by *MK-JR*. Let $\hat{\phi}^*$ 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 $\hat{C}_{max}^*$, for all job orders Φ . Particularly, $\check{C}_{max}^*$ is estimated with regard to $\check{\phi}^*$ by using Formula (3.3). $\hat{C}_{max}^*$ is estimated with regard to $\hat{\phi}^*$ with the Formula:

$$\hat{C}_{max}^* = \max_{1 \leq k \leq n} \left\{ \sum_{i=1}^k T_i^M + \max_{1 \leq i \leq n} \{\hat{t}_i^M\} + \sum_{i=k}^n T_i^R + \max_{1 \leq i \leq n} \{\hat{t}_i^R\} \right\}.$$

Proof Let ϕ denote an arbitrary job order. Our proof makes up of the following two parts:

(i). $C_{max} \geq \check{C}_{max}^*$.

Proof: Let's consider the following two scenarios:

(1). There is a scenario that the job execution order is consistent between the map phase and reduce phase for the generalized case, i.e, $c_j^{\mathcal{M}} \leq c_k^{\mathcal{M}}$ if J_j is submitted before J_k . We can construct a simplified case regarding ϕ with $(T_i^{\mathcal{M}}, T_i^{\mathcal{R}}) = (\frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|S^{\mathcal{M}}|}, \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|S^{\mathcal{R}}|})$ for each job J_i . Let $\check{C}_{max}$ denote the makespan for the simplified case. Let $\check{c}_i^{\mathcal{M}}$ and $\check{c}_i^{\mathcal{R}}$ denote the completion time at the map phase and reduce phase for the simplified case respectively. Then we have $\check{c}_i^{\mathcal{M}} = \frac{\sum_{1 \leq j \leq i} \sum_{k=1}^{|J_j^{\mathcal{M}}|} t_{j,k}^{\mathcal{M}}}{|S^{\mathcal{M}}|}$. However, for the generalized case (See Figure 3.3), due to the *non-divisible* condition for some jobs, we have $c_i^{\mathcal{M}} \geq \frac{\sum_{1 \leq j \leq i} \sum_{k=1}^{|J_j^{\mathcal{M}}|} t_{j,k}^{\mathcal{M}}}{|S^{\mathcal{M}}|}$ by considering the possible tail (e.g. a map task tail from 24 to 28 time unit for job J_4 in Figure 3.3). It reveals that for job J_i of the simplified case, its map tasks complete no later than those of the corresponding job of the generalized case at the map phase. Moreover, even though we let $\check{c}_i^{\mathcal{M}} = c_i^{\mathcal{M}}$ for each job J_i , we still have $\check{c}_i^{\mathcal{R}} \leq c_i^{\mathcal{R}}$ by considering the possible tail (e.g. a reduce task tail from 36 to 44 time unit for job J_4 in Figure 3.3) at the reduce phase. Note that $c_i = c_i^{\mathcal{R}}$ and $\check{c}_i = \check{c}_i^{\mathcal{R}}$, where $\check{c}_i$ denotes the completion time for job J_i of the simplified case. Therefore, we have $C_{max} \geq \check{C}_{max}$. On the other hand, we can obtain an optimal job order $\check{\phi}^*$ for makespan $\check{C}_{max}^*$ in the simplified case with *MK-JR*, in terms of *Johnson's Rule*. Then we have $\check{C}_{max} \geq \check{C}_{max}^*$. So it holds $C_{max} \geq \check{C}_{max}^*$.

(2). Besides, there is another scenario that the job execution is out of order between map phase and reduce phase for the generalized case (e.g. an out of order execution between J_1 and J_2 in the reduce phase in Figure 3.3). We can construct a simplified case based on the job execution order (denoted as ϕ') of the reduce phase of the generalized case, with $(T_i^{\mathcal{M}}, T_i^{\mathcal{R}}) = (\frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|S^{\mathcal{M}}|}, \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|S^{\mathcal{R}}|})$ for each job J_i . Let's consider an arbitrary pair of jobs (e.g., J_j, J_k) with different orders between ϕ (e.g. $J_j \rightarrow J_k$ in ϕ) and ϕ' (e.g. $J_k \rightarrow J_j$ in ϕ'). It can be noted that there must be a tail for J_j in the generalized case, as out of order condition. After we exchange their order (i.e. to be $J_k \rightarrow J_j$), we have $\check{c}_k^{\mathcal{M}} < c_k^{\mathcal{M}}$, based on the facts that: (a). the map tasks of J_k in the simplified case will begin earlier than in the generalized case; (b). the execution time for the map tasks of J_k in the simplified case will be $\frac{\sum_{j=1}^{|J_k^{\mathcal{R}}|} t_{k,j}^{\mathcal{R}}}{|S^{\mathcal{M}}|}$, which is smaller than that in the generalized case. Moreover, we have $\check{c}_j^{\mathcal{M}} < c_j^{\mathcal{M}}$ by considering that the tail available at the last map wave of J_j in the generalized case. All of these indicate that, each job J_i in the simplified case will complete its map tasks earlier than (or the same as) that in the generalized case. We therefore can guarantee $\check{c}_i^{\mathcal{R}} \geq c_i^{\mathcal{R}}$, for which the reason is the same as what we explained in case (1). Therefore, we have $C_{max} \geq \check{C}_{max} \geq \check{C}_{max}^*$.

By combining (1) and (2), we therefore have $C_{max} \geq \check{C}_{max}^*$, where $\check{C}_{max}^*$ can be estimated by using Formula (3.3) regarding $\check{\phi}^*$.

(ii). $C_{max} \leq \hat{C}_{max}^*$.

Proof:

We form $\hat{\phi}^*$ by reversing the result (denoted by ϕ_1) of *MK-JR*. In terms of Lemma 3.2, we have $C_{max} \leq \hat{C}_{max} = \max_{1 \leq k \leq n} \{ \sum_{i=1}^k T_i^{\mathcal{M}} + \max_{1 \leq i \leq k} \{\hat{t}_i^{\mathcal{M}}\} + \sum_{i=k}^n T_i^{\mathcal{R}} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\} \} \leq \max_{1 \leq k \leq n} \{ \sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^n T_i^{\mathcal{R}} \} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{M}}\} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\}$. Note that $\max_{1 \leq k \leq n} \{ \sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^n T_i^{\mathcal{R}} \} = \hat{C}_{max}^*$ is just the makespan estimation (Formula 3.3) for $\hat{\phi}^*$ in the simplified case. According to the *Johnson's Rule*, we know that ϕ_1 is the optimal job order for makespan in the simplified case. Moreover, in terms of Theorem 3.2, we can obtain that $\hat{\phi}^*$ is the worst-case job order for makespan in the simplified case. Thus we have $\hat{C}_{max}^*$ is maximum (i.e., $C_{max} \leq \hat{C}_{max}^*$) for any arbitrary job order ϕ .

Finally, our proof of Lemma 3.3 completes based on (i) and (ii). $\square$

A.3 Proof of Theorem 3.1

Theorem 3.1 *MK-JR* is an $(1 + \delta)$ -approximation algorithm for makespan optimization in the generalized case, where $\delta = \frac{\max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{M}}\} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\}}{\max_{1 \leq k \leq n} \{ \sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^n T_i^{\mathcal{R}} \}}, (0 \leq \delta \leq 1)$.

Proof Based on Lemma 3.2, it holds for the job order ϕ_1 produced by *MK-JR* that,

$$\begin{aligned} C_{max} &\leq \hat{C}_{max} = \max_{1 \leq k \leq n} \{ \sum_{i=1}^k T_i^{\mathcal{M}} + \max_{1 \leq i \leq k} \{\hat{t}_i^{\mathcal{M}}\} + \sum_{i=k}^n T_i^{\mathcal{R}} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\} \} \\ &\leq \max_{1 \leq k \leq n} \{ \sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^n T_i^{\mathcal{R}} \} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{M}}\} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\} \\ &= \check{C}_{max} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{M}}\} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\}. \end{aligned} \quad (A.1)$$

Let's define $\delta = \frac{\max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{M}}\} + \max_{1 \leq i \leq n} \{\hat{t}_i^{\mathcal{R}}\}}{\check{C}_{max}}, (0 \leq \delta \leq 1)$. Then Formula (A.1) is equivalent to

$$C_{max} \leq (1 + \delta) \cdot \check{C}_{max}. \quad (A.2)$$

Note that it holds $\check{C}_{max} = \check{C}_{max}^*$ for *MK-JR* based on *Johnson's Rule*. Moreover, according to Lemma 3.3, we have $\check{C}_{max}^* \leq C_{max}^{opt}$. Therefore, we have

$$C_{max} \leq (1 + \delta) \cdot C_{max}^{opt}. \quad (A.3)$$

$\square$

A.4 Proof of Theorem 3.2

Theorem 3.2 Suppose ϕ is the optimal job order whose makespan is C_{max}^{opt} , 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_{max}^{wst} , can be obtained by simply reversing ϕ .

Proof Consider Formula (3.3), it can be re-written as follows:

$$C_{max} = \max_{1 \leq k \leq n} \{T_k\}, \quad (A.4)$$

where

$$T_k = \sum_{i=1}^k T_i^{\mathcal{M}} + \sum_{i=k}^n T_i^{\mathcal{R}}.$$

Given a job order ϕ , let's define

$$F(\phi) = C_{max} = \max_{1 \leq k \leq n} \{T_k\}. \quad (A.5)$$

Therefore, our proof is equivalent to finding a job order ϕ^* such that $F(\phi^*) \geq F(\phi)$ for any ϕ .

Let's first consider a job order ϕ' constructed by interchanging J_j and J_{j+1} in ϕ . Then we have

$$F(\phi') = \max_{1 \leq k \leq n} \{T'_k\}. \quad (A.6)$$

where

$$T'_k = \sum_{i=1}^k T'_i^{\mathcal{M}} + \sum_{i=k}^n T'_i^{\mathcal{R}}$$

$$T'_i^{\mathcal{M}} = \begin{cases} T_i^{\mathcal{M}} & (i \neq j \wedge i \neq j+1) \\ T_{j+1}^{\mathcal{M}} & (i = j) \\ T_j^{\mathcal{M}} & (i = j+1) \end{cases}$$

$$T'_i^{\mathcal{R}} = \begin{cases} T_i^{\mathcal{R}} & (i \neq j \wedge i \neq j+1) \\ T_{j+1}^{\mathcal{R}} & (i = j) \\ T_j^{\mathcal{R}} & (i = j+1). \end{cases}$$

Then it holds

$$T'_k = T_k, \quad (k \neq j \wedge k \neq j+1). \quad (A.7)$$

Note that $F(\phi') \leq F(\phi)$ if $\max\{T_j, T_{j+1}\} \geq \max\{T'_j, T'_{j+1}\}$, otherwise $F(\phi') > F(\phi)$. To make $F(\phi^*) \geq F(\phi)$ be true for any ϕ , we only need to make sure

$$\max\{T_j, T_{j+1}\} \geq \max\{T'_j, T'_{j+1}\} \quad (A.8)$$

for all $1 \leq j < n$ in ϕ^* . Moreover, by subtracting $\sum_{i=1}^{j+1} T_i^{\mathcal{M}} + \sum_{i=j}^n T_i^{\mathcal{R}}$ from each term in Formula (A.8), it turns out to be

$$\max\{-T_{j+1}^{\mathcal{M}}, -T_j^{\mathcal{R}}\} \geq \max\{-T_j^{\mathcal{M}}, -T_{j+1}^{\mathcal{R}}\} \quad (\text{A.9})$$

or equivalent to

$$\min\{T_j^{\mathcal{M}}, T_{j+1}^{\mathcal{R}}\} \geq \min\{T_{j+1}^{\mathcal{M}}, T_j^{\mathcal{R}}\}. \quad (\text{A.10})$$

To satisfy Formula (A.10) for any two adjacent jobs J_j and J_{j+1} in ϕ^* , let's consider the conditions for the following possible cases:

- ①. When $T_j^{\mathcal{M}} \geq T_j^{\mathcal{R}}$ for Job J_j and $T_{j+1}^{\mathcal{M}} \leq T_{j+1}^{\mathcal{R}}$ for Job J_{j+1} , Formula (A.10) holds in this case without any further condition.
- ②. When $T_j^{\mathcal{M}} \leq T_j^{\mathcal{R}}$ for Job J_j and $T_{j+1}^{\mathcal{M}} \geq T_{j+1}^{\mathcal{R}}$ for Job J_{j+1} , we have $\min\{T_j^{\mathcal{M}}, T_{j+1}^{\mathcal{R}}\} \leq \min\{T_{j+1}^{\mathcal{M}}, T_j^{\mathcal{R}}\}$, which violates Formula (A.10). Hence this case is impossible in ϕ^* .
- ③. When $T_j^{\mathcal{M}} \geq T_j^{\mathcal{R}}$ for Job J_j and $T_{j+1}^{\mathcal{M}} \geq T_{j+1}^{\mathcal{R}}$ for Job J_{j+1} , to satisfy Formula (A.10), we should keep the following hold $T_j^{\mathcal{R}} \leq T_{j+1}^{\mathcal{R}}$.
- ④. When $T_j^{\mathcal{M}} \leq T_j^{\mathcal{R}}$ for Job J_j and $T_{j+1}^{\mathcal{M}} \leq T_{j+1}^{\mathcal{R}}$ for Job J_{j+1} , to satisfy Formula (A.10), we should guarantee $T_j^{\mathcal{M}} \geq T_{j+1}^{\mathcal{M}}$.

These give us guidance to find ϕ^* in the following way: Partition the jobs set J into two disjoint sub-sets J_A and J_B . Set J_A contains those jobs J_i whose $T_i^{\mathcal{M}} > T_i^{\mathcal{R}}$. Set J_B contains the remaining jobs. Sort jobs in J_A in non-decreasing order of $T_i^{\mathcal{R}}$, according to ③. Sort jobs in J_B in non-increasing order of $T_i^{\mathcal{M}}$, according to ④. Finally, ϕ^* is formed by appending the sorted J_B to the end of J_A according to ①②.

Note that the resultant ϕ^* of the above job ordering rule is just equivalent to reversing the order of the result based on *Johnson's Rule*. Hence Theorem 3.2 is true. $\square$

A.5 Proof of Theorem 3.3

Theorem 3.3 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.

Proof Let N denote the number of slave nodes. Let $|s^{\mathcal{M}}|$ and $|s^{\mathcal{R}}|$ denote the number of map slots and reduce slots configured per slave node respectively. We thereby have $|\mathcal{S}^{\mathcal{M}}| = N \cdot |s^{\mathcal{M}}|$ and $|\mathcal{S}^{\mathcal{R}}| = N \cdot |s^{\mathcal{R}}|$. Moreover, for each job J_i , we have $T_i^{\mathcal{M}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|} = \frac{1}{N} \cdot \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|}$ and $T_i^{\mathcal{R}} = \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|} = \frac{1}{N} \cdot \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|}$. Our proof is equivalent to validating that the rule steps of each algorithm are N -independent.

(i). ϕ_1 is N -independent for MK_JR .

Proof: For MK_JR , we only need to check its Step 2(a) and 2(b).

(1). *Step 2(a) N -independent Validation.* For each Job $J_i \in J$ in J_A , the condition $T_i^{\mathcal{M}} \leq T_i^{\mathcal{R}} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|} \leq \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} \leq \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|}$. Similarly, for each Job $J_i \in J$ in J_B , we have $T_i^{\mathcal{M}} > T_i^{\mathcal{R}} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} > \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|}$.

(2). *Step 2(b) N -independent Validation.* For each Job J_i in the sorted set J_A , it holds $T_i^{\mathcal{M}} \leq T_{i+1}^{\mathcal{M}} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} \leq \frac{\sum_{j=1}^{|J_{i+1}^{\mathcal{M}}|} t_{i+1,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} \leq \frac{\sum_{j=1}^{|J_{i+1}^{\mathcal{M}}|} t_{i+1,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|}$. Moreover, for each Job J_i in the sorted set J_B , it has $T_i^{\mathcal{R}} \geq T_{i+1}^{\mathcal{R}} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|} \geq \frac{\sum_{j=1}^{|J_{i+1}^{\mathcal{R}}|} t_{i+1,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|} \cdot t_{i+1}^{\mathcal{R}} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|} \geq \frac{\sum_{j=1}^{|J_{i+1}^{\mathcal{R}}|} t_{i+1,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|}$.

(ii). ϕ_2 is N -independent for MK_TCT_JR .

Proof: Note in Step 1 of MK_TCT_JR , $T_i = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|\mathcal{S}^{\mathcal{M}}|} + \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|\mathcal{S}^{\mathcal{R}}|} = \frac{1}{N} \cdot \left(\frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} + \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|} \right)$. Let $T'_i = \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} + \frac{\sum_{j=1}^{|J_i^{\mathcal{R}}|} t_{i,j}^{\mathcal{R}}}{|s^{\mathcal{R}}|}$, then it holds $T_i = \frac{1}{N} \cdot T'_i$. For MK_TCT_JR , we need to exam its rule steps 2, 3 as follows:

(1). *Step 2 N -independent Validation.* Note $T = (\prod_{1 \leq i \leq n} T_i)^{\frac{1}{n}} = \frac{1}{N} \cdot (\prod_{1 \leq i \leq n} T'_i)^{\frac{1}{n}}$. For each Job $J_i \in J$ in J'_A , the condition $T_i \leq T \Leftrightarrow \frac{1}{N} \cdot \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} \leq \frac{1}{N} \cdot (\prod_{1 \leq i \leq n} T'_i)^{\frac{1}{n}} \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} \leq (\prod_{1 \leq i \leq n} T'_i)^{\frac{1}{n}}$. Similarly, for each Job $J_i \in J$ in J'_B , we have $T_i > T \Leftrightarrow \frac{\sum_{j=1}^{|J_i^{\mathcal{M}}|} t_{i,j}^{\mathcal{M}}}{|s^{\mathcal{M}}|} > (\prod_{1 \leq i \leq n} T'_i)^{\frac{1}{n}}$.

(2). *Step 3 N -independent Validation.* Based on (i) that MK_JR is N -independent, therefore it holds that Step 3 is N -independent.

In summary, our proof completes in terms of (i) and (ii). $\square$