PERFORMANCE MODELING AND RESOURCE MANAGEMENT FOR MAPREDUCE APPLICATIONS

Zhuoyao Zhang

A DISSERTATION
in
Computer and Information Science
Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2014

Dedicated to my parents.

Acknowledgments

From the first day I joined the Ph.D program in Penn, I have spent five fruitful years in this group until finally, my research work comes into this dissertation. Here, I would like to express my deepest gratitude to my advisor, my collaborators, my friends and family and other individuals who support me through these years. Without their help, it is not possible for me to go this far.

First of all, I would like to thank my advisors, Professor Boon Thau Loo and Insup Lee who offered me the opportunity to pursue graduate studies at Penn and guided me through my research work. Insup provided me great support during my early years by exposing me to different research areas through our discussions. He also introduced me the work of real-time system which later helps me form the initial idea of my dissertation work. Boon have been extremely supportive through each step during my study. He gave me valuable advice on shaping my research directions, refining theoretical approaches as well as on technical details like designing experiments and analyzing results. He is also a great mentor who is open minded and willing to encourage students exploring new ideas, so that I could have freedom to choose research topics I am interested. I have learned from him not only the research skill, but also the importance of collaboration, passion about work, self-discipline and life balancing.

Next, I would like to thank Dr. Lucy Cherkasova who is my mentor during my internship in HP Labs. We worked together for two summers and have been kept collaborating remotely for more than two years. It was an exciting and fruitful working experience, during this time period, we have published papers on several conferences, workshops and journals. We filed two patents and presented our work in different occasions. All this could be achieved without her help. Lucy is especially patient and considerate in mentoring my research work as well as other aspects in my Ph.D life. I remember she walked through every page of my slides in preparing my talk and rehearsal multiple times with me even for internal presentations and that really helps me to clear my thoughts and is especially useful when I need to explain my works to other researchers later. The time we spent in HP Labs has been one of the most precious pieces of memory in my life.

I would also like to thank my collaborators. Among them, I would like to first thank Dr. Godfrey Tan who I worked with in my first project on job scheduling in large-scale distributed systems and encouraged me a lot during my first years as a Ph.D student. Dr. Linh T.X. Phan who is now a research assistant professor in Penn has a profound influence on my early research. She first brought the idea to introduce theories from real-time system into the parallel data processing platform which inspired me for my later works. I was also fortunate enough to collaborate with many other individuals from different places. Saumya Jain and Qi Zheng who were Penn graduate students and worked with me on several projects. I appreciate a lot for their help in system implementation and experiments. Abhishek Verma and Feng Yan are my collaborators during my summer intern and we worked together on MapReduce related problems from whom I have learnt a lot about Hadoop system. I also had a great time work with other students in the group such as Sanchit Aggarwal, Yang Li and Harjot Gill, who broadened my horizon, and enabled me to explore different types of problems outside my core research area.

This dissertation benefited from many great suggestions provided by my thesis committee members, professor Zachary Ives who is also my WPE-II committee chair, professor Susan Davidson and Andreas Haeberlen who provided me valuable advise through the write up.

In addition, I have to thank members from the NetDB group: Wenchao Zhou, Changbin Liu, Anduo Wang, Mengmeng Liu, Alex Gurney and Ling Ding, all of them have given me great suggestions in preparing my WPE-II writing and presentation, my thesis proposal defense and my job searching process. I also would like to give my special thanks to Mike Felker for helping me go through many tedious details of administrative tasks, and also to Cheryl Hickey and many other wonderful staff who make my life much easier.

Besides my research work, I want to thank my friends that colors my life after work. Special thanks to my best friend Lin Shao who listened to me and stayed with me through my hard times. My current and previous roommates: Qi Zhang, Jie Li, Xiang Yang, Weiyu Zhang, Yanfei Wang, Jiechang Hou, Matt Malloy, Hanjun Xiao, Yang Wu for those memorable times we shared together. Also friends I made these years: Zhuowei Bao, Chen Chen, Naobo Chen, Jian Chang, Qing Duan, Pengfei Huang, Zhihao Jiang, Dong Ling, Gang Song, Shaohui Wang, Meng Xu, Zhepeng Yan, Yifei Yuan, Mingchen Zhao, Jianzhou Zhao and many others.

My research was funded in part by the following funding agencies and sources: NSF CNS-1117185, CNS-1040672, CNS-0834524, CNS-0845552, IIS-0812270 and ARO grant W911NF-11-1-0403.

Last but not least, I dedicate this dissertation to my parents, Yongting Zhang and Xunjian Liu for their selfless love. They have made great sacrifice to support my study abroad and my dreams. In the past five years, I spent very little time with them which I believe has made their life much more difficult especially I am the only child in the family, but they never complied a word about that. It would worth everything if this dissertation could bring them proud and comfort.

ABSTRACT

PERFORMANCE MODELING AND RESOURCE MANAGEMENT FOR MAPREDUCE APPLICATIONS

Zhuoyao Zhang

Boon Thau Loo
Insup Lee

Big Data analytics is increasingly performed using the MapReduce paradigm and its open-source implementation Hadoop as a platform choice. Many applications associated with live business intelligence are written as complex data analysis programs defined by directed acyclic graphs of MapReduce jobs. An increasing number of these applications have additional requirements for completion time guarantees. The advent of cloud computing brings a competitive alternative solution for data analytic problems while it also introduces new challenges in provisioning clusters that provide best cost-performance trade-offs.

In this dissertation, we aim to develop a performance evaluation framework that enables automatic resource management for MapReduce applications in achieving different optimization goals. It consists of the following components: (1) a performance modeling framework that estimates the completion time of a given MapReduce application when executed on a Hadoop cluster according to its input data sets, the job settings and the amount of allocated resources for processing it; (2) a resource allocation strategy for deadline-driven MapReduce applications that automatically tailors and controls the resource allocation on a shared Hadoop cluster to different applications to achieve their (soft) deadlines; (3) a simulator-based solution to the resource provision problem in public cloud environment that guides the users to determine the types and amount of resources that should lease from the service provider for achieving different goals; (4) an optimization strategy to automatically determine the optimal job settings within a MapReduce application for efficient execution and resource usage. We validate the accuracy, efficiency, and performance benefits of the proposed framework using a set of realistic MapReduce applications on both private cluster and public cloud environment.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Data-intensive analytic applications have become core to the functions of the modern enterprises. Large companies like Google, Facebook, and LinkedIn are processing and analyzing Terabytes of data every day. These data analytic tasks range from business intelligent analytics [29], social network connection analysis [16], to more advanced scientific data analysis and machine learning applications [89] and the amount of data produced daily is exploding [15].

The enterprises and organizations are experiencing a paradigm shift towards large-scale data intensive computing. Many of them are increasingly using the MapReduce paradigm [21] and its open-source implementation Hadoop [72] as a platform choice for their *Big Data* analysis as it offers a simple and powerful framework for processing large data sets on distributed systems: the program logic is simply expressed by the *map* and *reduce* functions and the MapReduce execution engine will automatically execute the application in parallel on a set of nodes, coordinate their executions and handle failures transparently.

For more complex data analytics, several projects, such as Pig [26], Hive [63], Scope [17], and Dryad [32] provide high-level SQL-like abstractions on top of MapReduce engines to raise the level of abstraction for processing large data sets. These frameworks allow data analysts to specify complex analytic tasks without directly writing map and reduce functions and will compile the specified program automatically into *directed acyclic graphs* (DAGs) of MapReduce jobs which we referred to as MapReduce workflows.

1.1 Motivation

Though first introduced for batch-oriented workloads, there is recently an emerging technological trend to shift towards using MapReduce and the frameworks on top of it in support of *latency-sensitive* applications, e.g., personalized recommendations [19], advertisement placement [18], real-time web indexing [24] etc. These applications are typically a part of an elaborate business pipeline, and have to produce results by a certain deadline. There is a need for a mechanism that could automatically tailor and control resource allocations for different applications in shared MapReduce clusters to achieve their (soft) deadlines. Unfortunately, the existing Hadoop implementation does not support resource management for such latency sensitive applications.

On the other hand, the advent of cloud computing provides a new delivery model with virtually unlimited resources. It allows the customers to deploy their Hadoop clusters by leasing computing and storage resources offered by the cloud providers. One of the open questions in such environments is to determine the right resource provision strategy i.e. the choice and the amount of resources that a user should lease from the service providers to achieve performance goals for their MapReduce applications. Moreover, instead of a fixed capital investment that made up-front as for original private cluster, cloud providers offer a more cost-efficient option for many users in a ”pay-as-you-go” fashion, i.e., the customers only pay for the time they used the rented resources. As a result, the monetary cost for executing the user's workloads becomes another optimization goal which should be considered in making the cluster provision decision.

The solutions to the above problems rely on a profound understanding of the relationship between the execution performance of a given MapReduce application and the amount of resources available for processing it. Currently, there is a lack of performance models that could accurately predict the completion time for a given MapReduce application according to its input dataset(s), the job settings and the amount of allocated resources. It is especially challenging to develop such performance model because of nondeterminism during the execution due to the interference of applications in a shared cluster, the non-uniform data distribution and the heterogeneity of the hardware.

Based on the above motivation, the goal of this dissertation is therefore to **develop a performance evaluation framework that enables automatic resource management for MapReduce applications in achieving different optimization goals.** Towards developing such performance evaluation framework, we need to address the following challenges:

- **Automation:** The desired framework should automatically control the resource management with minimal manual interventions and work seamlessly with the Hadoop distribution.
- **Accuracy:** The desired framework should be sufficiently accurate in predicting the application completion time or required amount of resources even with the presence of system nondeterminism.
- **Efficiency:** The desired framework should be efficient and able to provide timely response in support of latency sensitive applications in online environment.
- **Generality:** The desired framework should be applied to different applications, e.g., complex MapReduce workflows and Hadoop clusters, e.g., heterogeneous cluster that consists of different type of hardwares.

1.2 Contributions of the dissertation

In this dissertation, we aim to develop a performance evaluation framework that enables automatic resource management for MapReduce applications in achieving different optimization goals. The techniques we use combine of mathematical analysis, benchmarking, simulation, implementation, deployments and empirical measurements on both private cluster and public cloud platform. We focus on Hadoop – the most widely used open source implementation of MapReduce platform and the Pig framework [26] – a popular and widely-adopted system built on

Hadoop for expressing a broad variety of data analysis tasks to build our model. While the results provided in this dissertation are based on the Hadoop and Pig experience, we believe that the proposed models and resource management strategies are general and can be applied for application executed on similar frameworks such as Hive and Dryad.

A more detailed description about the contributions of this dissertation is presented as follows.

1.2.1 Contribution 1: performance modeling framework for MapReduce applications

Problem: To enable resource management for MapReduce applications, the first challenging problem is to understand the relationship between the performance and the amount of resource available for executing the application. It requires a performance model which is able to estimate the completion time of a given MapReduce application according to different amount of computing resources. The estimated completion time also depends on the performance of the platform hardware, the input data sets, the job settings and could be affected by the performance uncertainty caused by resource contention on a shared cluster. While there have been some research efforts [45, 28, 64, 66] towards developing performance models for MapReduce jobs, these techniques either rely on simplified assumptions or do not apply to complex applications that are expressed as a DAG of MapReduce jobs.

Solution: To solve the problem, we build a performance modeling framework [82, 86, 85, 88] which can accurately predict the completion time of a given application that consists of single or a DAG of MapReduce jobs according to its input dataset(s), the job settings and the amount of allocated resources. It consists of an ensemble of performance models that orchestrate the performance prediction at different system and applications levels which are:

- A *platform performance model* that estimates a phase duration as a function of processed data at the Hadoop level.
- A *MapReduce job model* that is used to predict a single MapReduce job execution time as a function of allocated resources.
- A *workflow performance model* that combines all parts together for evaluating the completion time of complex application which represented as a DAG of MapReduce jobs.

1.2.2 Contribution 2: resource allocation for deadline-driven MapReduce applications

Problem: In the enterprise setting, users would benefit from sharing Hadoop clusters and consolidating diverse applications over the same datasets. With the trend towards using MapReduce for latency sensitive applications, there is a challenge to automatically tailor and control resource allocations on such shared clusters for different applications to achieve their (soft) deadlines. The existing Hadoop implementation does not support resource management for those applications: The original scheduler employed in Hadoop is a simple FIFO scheduling policy that assigns all available resource to each job according to their submission time. The Fair scheduler [8] and Capacity scheduler [2] introduced later try to share the cluster resource among the running jobs either according to the job size or partitioned resource pool, but none of them aims to provide resource allocation in order to satisfy the completion time requirement for the applications.

Solution: We solve this problem using our deadline-driven resource allocation strategy based on our performance modeling framework [87, 85, 88]. Once we are able to estimate the application completion time according to the resource allocation, we could also solve the related problem that is to estimate the appropriate amount of resources required for completing an application with a given (*soft*) deadline. We first propose a simple *basic* approach which works efficiently for applications with single or sequential jobs, and then propose a *refined* approach that works for more complex applications that contain both sequential and concurrent jobs. Towards solving this problem, we also optimize an application execution by enforcing the optimal schedule of its concurrent jobs. Such optimization helps reducing the total completion time, and more importantly, it eliminates possible non-determinism of concurrent jobs' execution in the workflow, and therefore, enables a more accurate performance and resource requirement prediction.

1.2.3 Contribution 3: resource provision in public cloud environment

Problem: In contract to the private cluster, the advent of cloud computing provides an attractive alternative option to customers for provisioning a suitable size Hadoop cluster, consuming resources as a service, executing the MapReduce workload, and paying for the time these resources were used. One of the open questions in such environments is the choice of types and amount of resources that a user should lease from the service provider for optimizing both the performance and cost objectives. Specifically, the problem we are trying to solve in the dissertation is: given a workload, determine a Hadoop cluster(s) configuration (i.e., the number and types of VMs, and the job schedule) that provides best cost/performance trade-offs: i) minimizing the cost (budget) while achieving a given makespan target, or ii) minimizing the achievable jobs makespan for a given budget.

Solution: We offer a simulation-based framework for solving this problem [84, 83]. We first extract job profiles by executing the workloads on different (interested) platforms. Then, for each platform and a given Hadoop cluster size, we determine the optimized MapReduce job schedule i.e., the execution order of the jobs in the workloads. After that, our event based MapReduce simulator will take the job profiles and the schedules as inputs and output the simulated makespan/costs for executing the workloads. We first provide our solution for homogeneous clusters by iterating through all the possible choices to determine the optimal one. We then extend the approach for a heterogeneous solution that consists of sub-clusters of different types.

1.2.4 Contribution 4: performance optimization for MapReduce applications

Problem: Optimizing the execution efficiency of MapReduce jobs is an open challenge and has been studied from different perspectives [75, 28, 46, 71, **?**, 79]. In this dissertation, we focus on improving the execution performance for MapReduce applications by automatically tuning the job settings. i.e, the number of reduce tasks in each job as such parameter could significantly impact the total completion time as well as the resource usage. Determining the right number of reduce tasks is non-trivial: it depends on the input sizes of the job, on the Hadoop cluster size, and the amount of resources available for processing this job. This problem is more complicated for applications defined as MapReduce workflows given the data dependency of the jobs: the output of the previous job becomes the input of the next job, as a result, the job settings of the previous job could also have an impact on the map execution of the next job so as the entire completion time. Currently, it is solely the user's responsibility to configure the number of reduce tasks for each MapReduce job within the application. Such manual, experience based configuration probably leads to inefficient results.

Solution: Based on our performance modeling framework, we provide an automatic way for guiding the user efforts of tuning the reduce task settings in a MapReduce application while achieving different performance objectives [80, 81]. It contains two strategies for analyzing the performance trade-offs, i.e., to optimize the completion time while minimize the resource usage for its execution: a *local* optimization that searches for trade-offs at a job level, and a *global* optimization that makes the trade-off decisions at the workflow level.

1.3 Overview of dissertation

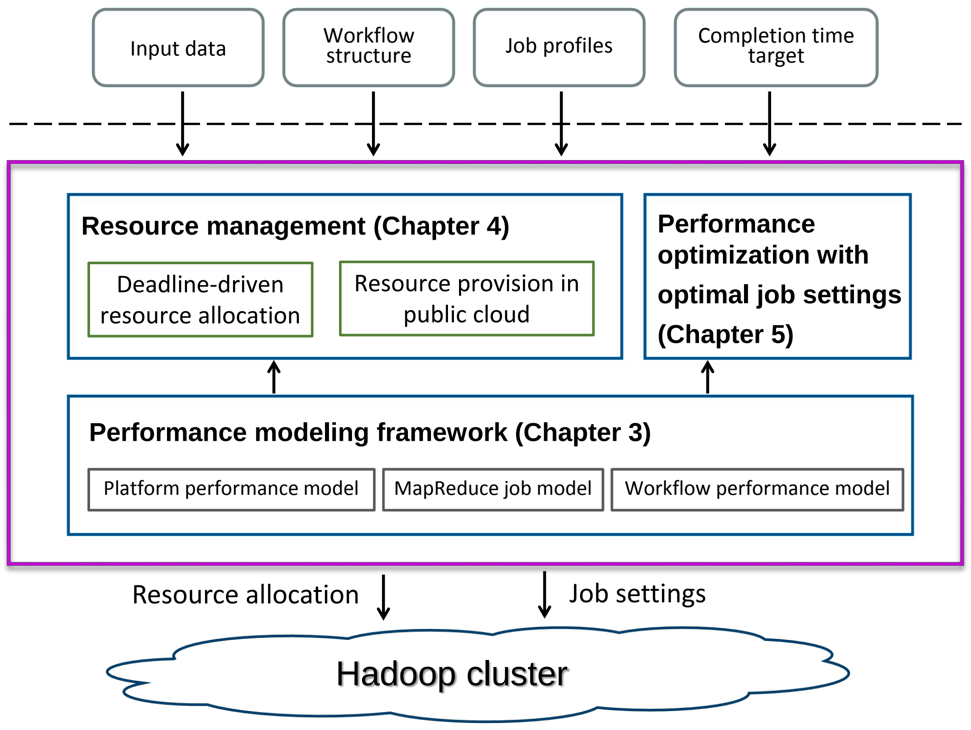

Input data
Workflow structure
Job profiles
Completion time target
Resource management (Chapter 4)
Deadline-driven resource allocation
Resource provision in public cloud
Performance optimization with optimal job settings (Chapter 5)
Performance modeling framework (Chapter 3)
Platform performance model
MapReduce job model
Workflow performance model
Resource allocation
Job settings
Hadoop cluster

Figure 1.1: Overview of the dissertation

Figure 1.1 presents an overview of our solution framework which shows the different components within the system and their connections. The rest of this dissertation is organized as follows: we first start by introducing the background on the MapReduce framework, its open source implementation Hadoop and the Pig system built on top of it in Chapter 2.

In Chapter 3, we introduce the performance modeling framework with a detailed description on each consisting performance model and demonstrate that the framework can accurately estimate the completion time of a given MapReduce application according to its input data sets and the amount of allocated resources.

In Chapter 4, we propose our resource management solutions by first introducing our resource allocation strategy for supporting multiple latency sensitive MapReduce applications executed on a shared Hadoop cluster. After that, we provide our resource provisioning strategy for guiding the user to select the platform from public cloud providers that provides the best cost/performance trade-offs for a given MapReduce workload.

In Chapter 5, we introduce our optimization strategy for MapReduce applications through tuning the job settings for better performance and resource usage. We propose both a local and a global optimization algorithms that applies on the job and workflow level respectively.

Chapter 6 describes the related work on performance modeling, resource management and optimization for MapReduce related platforms. Chapter 7 summaries the dissertation work and proposes a few directions for future research work.

Chapter 2

Background

This chapter provides a basic background on the MapReduce framework [20, 21] and its open source implementation Hadoop [9, 72] as well as a framework built on top of it: the Pig system [26] that offers a higher-level abstraction for expressing more complex analytic tasks using SQL-style constructs.

2.1 MapReduce framework

The MapReduce framework was first introduced by Google [21] and is now widely used in large-scale data processing on distributed clusters. In the MapReduce model, computation is expressed as two functions: map and reduce. The map function takes an input pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key k_2 are grouped together and then passed to the reduce function. The reduce function takes intermediate key k_2 with a list of values and processes them to form a new list of values.

$$\begin{aligned} map(k_1, v_1) &\rightarrow list(k_2, v_2) \\ reduce(k_2, list(v_2)) &\rightarrow list(v_3) \end{aligned}$$

MapReduce jobs are executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*. The underlying system automatically execute these map and reduce tasks in parallel on distributed clusters. The scheduling of tasks in MapReduce is performed by a master node which manages a number of worker nodes in the cluster.

Figure 2.1 shows a high level description of the MapReduce architecture. In the map stage, each map task processes a logical split of input data (typically stored in a distributed file system such as HDFS), applies the user-defined map function, and generates the intermediate set of key/value pairs. In the reduce stage, each reduce task fetches its partition of intermediate key/value pairs from all the map tasks and merges the data with the same key. After that, it applies the user-defined reduce function to produce the aggregate values and then write the results back to HDFS.

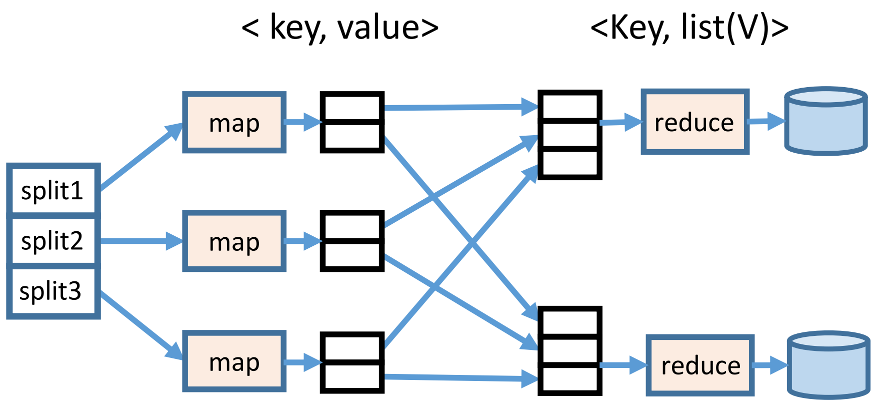


Figure 2.1: Architecture of MapReduce framework

Hadoop [9, 72] is an open source implementation of the MapReduce framework and has been widely used in many companies such as Yahoo!, Facebook, LinkedIn etc. In Hadoop, the cluster is constructed to contain a master node called the JobTracker while the other node are called worker node. Each worker node in the cluster is configured with a fixed number of map and reduce *slots* which represent the resource unit for processing the map and reduce tasks.

The worker nodes periodically connect to the JobTracker to report its current status and the available slots. The JobTracker decides the next job to execute based on the reported information and according to a scheduling policy. The popular job schedulers include FIFO, Fair [77], and Capacity scheduler [2]. The assignment of tasks to slots is done in a greedy way: assign a task from the selected job immediately whenever a worker reports to have a free slot. If the number of tasks belonging to a MapReduce job is greater than the total number of slots available for processing the job, the task assignment will take multiple rounds, which we call *waves*.

Optionally, a Hadoop job could define a combiner function that aggregates the map outputs. It takes a key and a subset of associated values and produces a single value. The combiner function is useful when it efficiently reduces the amount of data that need to be transferred to the reduce tasks.

The Hadoop implementation also includes *counters* for recording timing information such as start and finish timestamps of the tasks or the number of bytes read and written by each task. These counters are sent by the worker nodes to the master periodically with each heartbeat and are written to logs after the job is completed.

2.2 Pig programs

The Hadoop Pig system [26] which is among the similar projects such as Hive [63], Scope [17], and Dryad [32], aims to raise the level of abstraction for processing large datasets using Hadoop. It provides high-level SQL-like abstractions on top of MapReduce engines that enable data analysts to specify complex analytics tasks without directly writing Map and Reduce functions. The current Pig system is made up of the following two main components:

- The *language*, called Pig Latin, that combines high-level declarative style of SQL and the low-level procedural programming of MapReduce. A Pig program is similar to specifying a query execution plan: it represent a sequence of steps, where each one carries a single data transformation using a high-level data manipulation constructs, like *filter*, *group*, *join*, etc. In this way, the Pig program encodes a set of explicit dataflows.

- The *execution environment* to run Pig programs. The Pig system takes a Pig Latin program as input, compiles it into a DAG of MapReduce jobs, and coordinates their execution on a given Hadoop cluster. Pig relies on underlying Hadoop execution engine for scalability and fault-tolerance properties.

The following specification shows a simple example of a Pig program. It describes a task that operates over a table *URLs* that stores data with the three attributes: `(url, category, pagerank)`. This program identifies for each `category` the `url` with the highest `pagerank` in that `category`.

```
URLs = load 'dataset' as (url, category, pagerank);
groups = group URLs by category;
result = foreach groups generate group, max(URLs.pagerank);
store result into 'myOutput'
```

The example Pig program is compiled into a single MapReduce job. Typically, Pig programs are more complex, and can be compiled into an execution plan consisting of several stages of MapReduce jobs, some of which can run concurrently. The structure of the execution plan can be represented by a DAG of MapReduce jobs that could contain both concurrent and sequential branches. Figure 2.2 shows a possible DAG of five MapReduce jobs $\{j_1, j_2, j_3, j_4, j_5\}$, where each node represents a MapReduce job, and the edges between the nodes represent *data dependencies* between jobs.

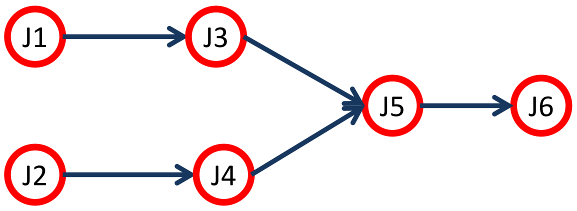


Figure 2.2: Example of a Pig program' execution plan represented as a DAG of MapReduce jobs.

To execute the plan, the Pig engine first submits all the *ready* jobs (i.e., the jobs that do not have data dependencies on the other jobs) to Hadoop. After Hadoop has processed these jobs, the Pig system deletes them and the corresponding edges from the processing DAG, and identifies and submits the next set of ready jobs. This process continues until all the jobs are completed. In this way, the Pig engine partitions the DAG into multiple stages, each containing one or more independent MapReduce jobs that can be executed concurrently. Note that for stages with concurrent jobs, there is no specifically defined ordering in which the jobs are going to be executed by Hadoop. For example, the DAG shown in Figure 2.2 is partitioned into the following four stages for processing:

- first stage: $\{j_1, j_2\}$;
- second stage: $\{j_3, j_4\}$;
- third stage: $\{j_5\}$;
- fourth stage: $\{j_6\}$.

Chapter 3

Performance Modeling Framework

In this chapter, we introduce a performance modeling framework that aims to estimate the MapReduce application completion time as a function of the allocated resource, the input data sets and the job settings. The intuition of our work comes from two parts:

- We observe that the executions of map and reduce tasks consist of specific, well-defined data processing phases. Only map and reduce functions are custom and their computations are user-defined for different MapReduce jobs. The executions of the remaining phases are generic, i.e., the logic of these phases is defined by the Hadoop processing framework. The execution time of each generic phase depends mostly on the amount of data processed by the phase and the I/O performance of underlying Hadoop cluster.
- In MapReduce environments, many production jobs are run periodically on new data. For example, Facebook, Yahoo!, and eBay process terabytes of data and event logs per day on their Hadoop clusters for spam detection, business intelligence and different types of optimization. We can extract a representative job profile that reflects the performance characteristics of the customized map and reduce functions and use the job profiles to predict the future execution of the same applications when executed on a different set of input data.

Specifically, the modeling framework consists of *three* performance models that orchestrates the prediction of the application completion time at different system and applications levels. Figure 3.1 outlines the ensemble of performance models designed for evaluating the application completion time.

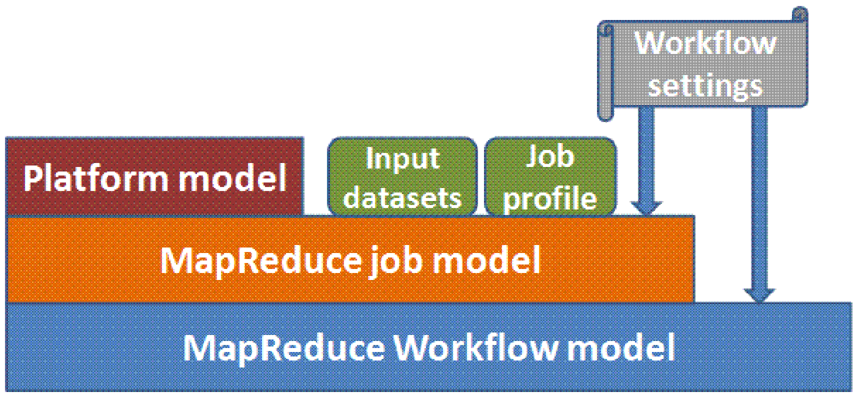


Figure 3.1: Ensemble of Models.

Chapter 3.1 first describes a platform performance model that estimates a generic phase duration as a function of processed data on a given Hadoop cluster. With the information of the input dataset(s) and the job settings, we could estimate the amount of data processed by each job (and the tasks within the job) in the application. Based on the estimated data flowing through each phases, we are able to predict the duration for generic phases by applying the derived platform performance model. To estimate the duration of the customized map and reduce phases, we extracted a compact job profile that captures the performance of the map and reduce functions and use it to approximate their durations according to the number of records processed by those functions. Once we estimated the duration of all phases, the map and reduce task durations can be estimated as the sum of the phase durations that belong to the task.

After that, Chapter 3.2 presents a MapReduce job model that is used to predict execution time of MapReduce application that contains single job according to the map (reduce) task durations and the amount of allocated resources (i.e., number of map and reduce slots). Finally, Chapter 3.3 present the workflow performance model that combines all parts together for evaluating the completion time for complex MapReduce workflows.

3.1 Platform performance model

We first describe our benchmarking approach for building a MapReduce platform model that aims to predicting the completion time of different MapReduce phases as a function of processed data. We use a set of microbenchmarks to profile generic phases of the MapReduce processing pipeline of a small given Hadoop cluster. Based on the benchmark results, we then derive an accurate platform performance model of a given cluster. The advantage of our approach includes

- Generality: The platform performance model is derived *once* for the given cluster, and then can be reused for characterizing performance of generic phases of different applications.
- Scalability: We derive the model using a small test cluster and then use it for the larger production cluster with the same hardware. The benchmarking process is performed without interfering the production cluster.

3.1.1 Profiling MapReduce phases

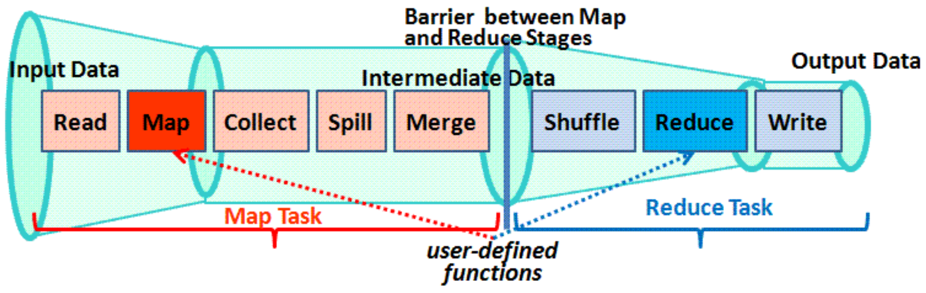


Figure 3.2: MapReduce Processing Pipeline.

As showed in Figure 3.2, the execution of each map (reduce) task is comprised of a specific, well-defined sequence of processing phases.

For each map task, it first reads a split of the input data from the Hadoop distributed file system (HDFS) (*read* phase), applies the user-defined map function, and generates the intermediate set of key/value pairs (*read* phase). The map task then buffers the map outputs and sorts and spills the data into disk once the intermediate data grows beyond certain threshold (*collect* and *spill* phase). After all the input data are processed, the map task merges the spilled data and partitions them for different reduce tasks according to a partition function (*merge* phase).

For each reduce task, it first fetches its partition of intermediate key/value pairs from all the map tasks and sort/merges the data with the same key (*shuffle* phase). After that, it applies the user-defined reduce function to the merged value list to produce the aggregate results (*reduce* phase). Finally, the reduce outputs are written back to HDFS (*write* phase).

Note, that only map and reduce phases with customized map and reduce functions execute the user-defined pieces of code. The execution of the remaining phases are *generic* (i.e., the logic of these phases is defined by Hadoop code), and their durations depend mostly on the amount of data flowing through a phase and the I/O performance of the underlying Hadoop cluster. Our goal is therefore to derive a *platform performance model* that predicts a duration of each generic phase on a given Hadoop cluster platform as a function of processed data.

In order to accomplish this, we run a set of microbenchmarks that create different amounts of data for processing per map (reduce) tasks and for processing by their phases. We profile the duration of each generic phase during the task execution and derive a function that defines a phase performance as a function of the processed data from the collected measurements.

For map tasks, we profile the following generic phases:

- *Read* phase – a map task typically reads a block (e.g., 64 MB) from the Hadoop distributed file system (HDFS). However, written data files might be of arbitrary size, e.g., 70 MB. In this case, there will be two blocks: one of 64 MB and the second of 6 MB, and therefore, map tasks might read files of varying sizes. We measure the duration of the read phase as well as the amount of data read by the map task.

- *Collect* phase – this generic phase follows the execution of the map phase with a user-defined map function. We measure the time it takes to buffer map phase outputs into memory and the amount of generated intermediate data.
- *Spill* phase – we measure the time taken to locally sort the intermediate data and partition them for the different reduce tasks, applying the combiner if available, and then writing the intermediate data to local disk.
- *Merge* phase – we measure the time for merging different spill files into a single spill file for each destined reduce task.

For reduce tasks, we profile the following generic phases:

- *Shuffle* phase – we measure the time taken to transfer intermediate data from map tasks to the reduce tasks and merge-sort them together. We combine the shuffle and sort phases because in the Hadoop implementation, these two sub-phases are interleaved. The processing time of this phase depends on the amount of intermediate data destined for each reduce task and the Hadoop configuration parameters. In our testbed, each JVM (i.e., a map/reduce s-lot) is allocated 700 MB of RAM. Hadoop sets a limit (∼46% of the allocated memory) for in-memory sort buffer. The portions of shuffled data are merge-sorted in memory, and a spill file (∼320 MB) is written to disk. After all the data is shuffled, Hadoop merge-sorts first 10 spilled files and writes them in the new sorted file. Then it merge-sorts next 10 files and writes them in the next new sorted file. At the end, it merge-sorts these new sorted files. Thus, we can expect a different scaling function for a duration of the shuffle phase when the size of intermediate data per reduce task is larger than *3.2 GB* in our Hadoop cluster as the merge-sorts process need to scan the entire output data multiple times when merging more than 10 on disk files. For a different

Hadoop cluster, this threshold can be similarly determined from the cluster configuration parameters.

- *Write* phase – this phase follows the execution of the reduce phase that executes a custom reduce function. We measure the amount of time taken to write the reduce output to HDFS.

Note, that in platform profiling we do not include phases with user-defined map and reduce functions. However, we do need to profile these custom map and reduce phases for modeling the execution of given MapReduce applications:

- *Map* (*Reduce*) phase – we measure a duration of the entire map (reduce) function and the number of processed records. We normalize this execution time to estimate a processing time per record.

Apart from the phases described above, each executed task has a constant overhead for setting and cleaning up. We account for these overheads separately for each task.

For accurate performance modeling, it is desirable to minimize the overheads introduced by the additional monitoring and profiling technique. There are two different approaches for implementing phase profiling.

- *Counter based profiling:* The current Hadoop implementation includes several *counters* to record information such as the number of bytes read and written. We modified the Hadoop code by adding counters that measure durations of the six generic phases to the existing counter reporting mechanism. We can activate the subset of desirable counters in the Hadoop configuration for collecting the set of required measurements.
- *Dynamic instrumentation based profiling:* We also implemented the alternative profiling tool inspired by Starfish [28] approach based on *BTrace* – a dynamic instrumentation tool for Java [7]. This approach does have a special appeal for production Hadoop clusters because it has a zero overhead when monitoring is turned off. However, in general, the dynamic instrumentation

overhead is significantly higher compared to adding new Hadoop counters directly in the source code.

For building the platform performance mode, we execute a set of microbenchmarks (described in Chapter 3.1.2) and measure the durations of six *generic* execution phases for processing different amount of data: *read, collect, spill,* and *merge* phases for the map task execution, and *shuffle* and *write* phases in the reduce task processing. This profiling is done on a small test cluster (5-nodes in our experiments) with the same hardware and configuration as the production cluster. While for these experiments both profiling approaches can be used, the Hadoop counter-based approach is preferable due to its simplicity and low overhead, and that the modified Hadoop version can be easily deployed in this test environment.

For predicting the completion time for a particular MapReduce job, we needs additional measurements that characterize the execution of user-defined map and reduce functions of a given job. For profiling the map and reduce phases of the given MapReduce jobs in the production cluster we apply our alternative profiling tool that is based on *BTrace* approach. Remember, this approach does not require Hadoop or application changes, and can be switched on for profiling a targeted MapReduce job of interest. Since we only profile map and reduce phase executions the extra overhead is relatively small.

3.1.2 Microbenchmarks

We generate and perform a set of parameterizable microbenchmarks to characterize execution times of generic phases for processing different data amounts on a given Hadoop cluster by varying the following parameters:

- *Input size per map task* (M^{inp}): This parameter controls the size of the input read by each map task. Therefore, it helps to profile the *Read* phase durations for processing different amount of data.
- *Map selectivity* (M^{sel}): this parameter defines the ratio of the map output to the map input. It controls the amount of data produced as the output of the

map function, and therefore directly affects the *Collect*, *Spill* and *Merge* phase durations in the map task. Map output determines the overall amount of data produced for processing by the reduce tasks, and therefore impacting the amount of data proceeded by *Shuffle* and *Reduce* phases and their durations.

- *Number of map tasks* N^{map}: This parameter helps to expedite generating the large amount of intermediate data per reduce task.
- *Number of reduce tasks* N^{red}: This parameter helps to control the number of reduce tasks to expedite the training set generation with the large amount of intermediate data per reduce task.

Thus, each microbenchmark MB_i is parameterized as

$$MB_i = (M_i^{inp}, M_i^{sel}, N_i^{map}, N_i^{red}).$$

Each created benchmark uses input data consisting of 100 byte key/value pairs generated with *TeraGen* [1], a Hadoop utility for generating synthetic data. The map function simply emits the input records according to the specified map selectivity for this benchmark. The reduce function is defined as the identity function. Most of our benchmarks consist of a specified (fixed) number of map and reduce tasks. For example, we generate benchmarks with 40 map and 40 reduce tasks each for execution in our small cluster deployments with 5 worker nodes. We run benchmarks with the following parameters: M^{inp}={2MB, 4MB, 8MB, 16MB, 32MB, 64MB}; M^{sel}={0.2, 0.6, 1.0, 1.4, 1.8}. For each value of M^{inp} and M^{sel}, a new benchmark is executed. We also use benchmarks that generate special ranges of intermediate data per reduce task for accurate characterization of the shuffle phase. These benchmarks are defined by N^{map}={20,30,...,150,160}; $M^{inp} = 64MB$, $M^{sel} = 5.0$ and $N^{red} = 5$ which result in different intermediate data size per reduce tasks ranging from 1 GB to 12 GB.

We generate the *platform profile* by running a set of our microbenchmarks on the small 5-node test cluster that is similar to a given production Hadoop cluster. We gather durations of generic phases and the amount of processed data for all executed map and reduce tasks. A set of these measurements defines the *p*latform profile that is later used as the training data for a *p*latform performance model:

- *Map task processing:* in the collected platform profiles, we denote the measurements for phase durations and the amount of processed data for *read, collect, spill,* and *merge* phases as $(Dur_1, Data_1)$, $(Dur_2, Data_2)$, $(Dur_3, Data_3)$, and $(Dur_4, Data_4)$ respectively.
- *Reduce task processing:* in the collected platform profiles, we denote phase durations and the amount of processed data for *shuffle* and *write* as $(Dur_5, Data_5)$ and $(Dur_6, Data_6)$ respectively.

Figure 3.3 shows a small fragment of a collected platform profile as a result of executing the microbenchmarking set. There are six tables in the platform profile, one for each phase. Figure 3.3 shows fragments for *read* and *collect* phases. There are multiple map and reduce tasks that process the same amount of data in each microbenchmark. This is why there are multiple measurements in the profile for processing the same data amount.

Row *n*umber j	Data *M*B $Data_1$	*R*ead *m*sec Dur_1
1	16	2010
2	16	2020
...	...	...

Row *n*umber j	Data *M*B $Data_2$	Collect *m*sec Dur_2
1	8	1210
2	8	1350
...	...	...

Figure 3.3: A fragment of a platform profile for *read* and *collect* phases.

3.1.3 Platform modeling

Now, we describe how to create a *platform performance model* M_{Phases} which characterizes the phase execution as a function of processed data. To accomplish this goal, we need to find the relationships between the amount of processed data and durations of different execution phases using the set of collected measurements. Therefore, we build *six* submodels $M_1, M_2, ..., M_5$, and M_6 that define the relationships for *read, collect, spill, merge, shuffle,* and *write* respectively of a given Hadoop cluster. To derive these submodels, we use the collected platform profile (see Figure 3.3).

Below, we explain how to build a submodel M_i, where $1 \leq i \leq 6$. By using measurements from the collected platform profiles, we form a set of equations which express a phase duration as a linear function of processed data. Let $Data_i^j$ be the amount of processed data in the row j of platform profile with K rows. Let Dur_i^j be the duration of the corresponding phase in the same row j. Then, using linear regression, we solve the following sets of equations (for each $i = 1, 2, \cdots, 6$):

$$A_i + B_i \cdot Data_i^j = Dur_i^j, \;\; where \;\; j = 1, 2, \cdots, K \tag{3.1}$$

To solve for (A_i, B_i), one can choose a regression method from a variety of known methods in the literature (a popular method for solving such a set of equations is a non-negative Least Squares Regression). With ordinary least squares regression, a few bad outliers can significantly impact the model accuracy, because it is based on minimizing the overall absolute error across multiple equations in the set. To decrease the impact of occasional bad measurements and to improve the overall model accuracy, we employ robust linear regression [30]. (which is typically used to avoid a negative impact of a small number of outliers).

Let $(\hat{A}_i, \hat{B}_i)$ denote a solution for the equation set (1). Then $M_i = (\hat{A}_i, \hat{B}_i)$ is the submodel that defines the duration of execution phase i as a function of processed data. The *p*latform performance model is $M_{Phases} = (M_1, M_2, ..., M_5, M_6)$.

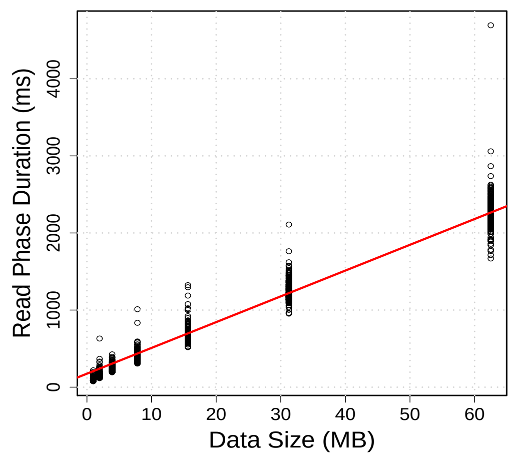
For the shuffle phase, according to the discussion in Chapter 3.1.1, we expect that there will be different behavior when it processes data smaller/larger than around 3.2 GB and thus better be approximated by a piece-wise linear function comprised of two linear functions: one for processing up to 3.2 GB of intermediate data per reduce task, and the second segment for processing the datasets larger than 3.2 GB.

We derived the platform performance model by executing the set of our microbenchmarks on a small 5-nodes clusters and collecting the corresponding phase durations. Each machine has four AMD 2.39GHz cores, 8 GB RAM and two 160G-

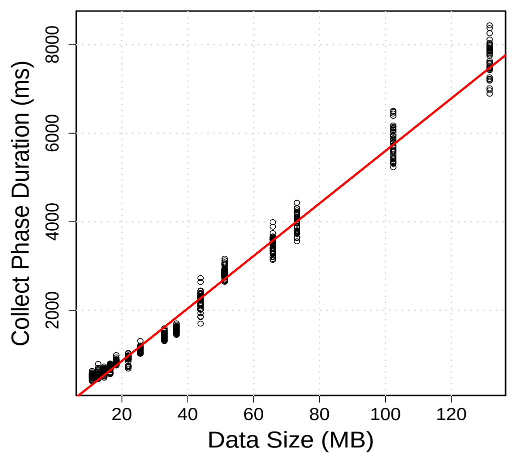
B hard disks. We used Hadoop 0.20.2 with additional two machines dedicated as the JobTracker and the NameNode. Each working node is configured with 2 map and 1 reduce slots. The file system block size is set to 64MB. The replication level is set to 3. We disabled speculative execution since it did not lead to significant improvements in our experiments.

Figure 3.4 shows the relationships between the amount of processed data and the execution durations of different phases for a given Hadoop cluster. It reflects the platform profile for six generic execution phases: *read*, *collect*, *spill*, and *merge* phases of the map task execution, and *shuffle* and *write* phases in the reduce task. Each graph has a collection of dots that represent phase duration measurements (Y-axes) of the profiled map (reduce) tasks as a function of processed data (X-axes). The red line on the graph shows the linear regression solution that serves as a model for the phase. As we can see (visually) the linear regression provides a good solution for five out of six phases. As it was expected, the shuffle phase is better approximated by a linear piece-wise function comprised of two linear functions.

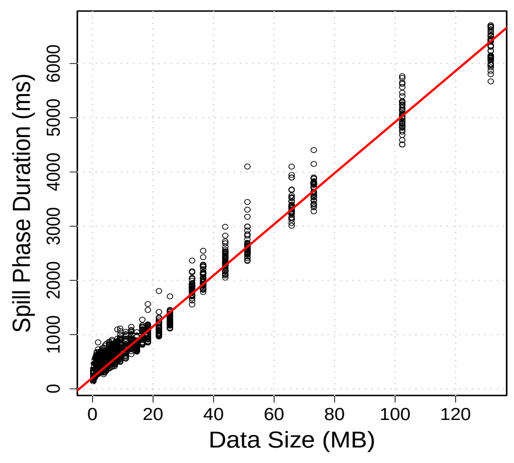
To validate whether our explanation on the shuffle phase behavior is correct, we perform a set of additional experiments. We configured each JVM (i.e., a map/reduce slot) with 2 GB RAM (compared to JVM with 700 MB of RAM used in previous experiments). As we explained earlier, Hadoop sets a limit ($\sim$46% of the allocated memory) for in-memory sort buffer. The portions of shuffled data are merge-sorted in memory, and a spill file (in the new case, $\sim$900 MB) is written to disk. After all the data is shuffled, Hadoop merge-sorts first 10 spilled files and writes them in the new sorted file. Then it merge-sorts next 10 files and writes them in the next new sorted file. Finally, at the end, it merge-sorts these new sorted files. Thus, we can expect that in the new configuration the shuffle performance is significantly different for processing intermediate data large than 9 GB. Figure 3.5 indeed confirms our conjecture: shuffle performance changes for processing intermediate data large than 9 GB. Indeed, the shuffle phase performance is affected by the JVM memory size settings, and its performance can be more accurately approximated by a linear piece-wise function.



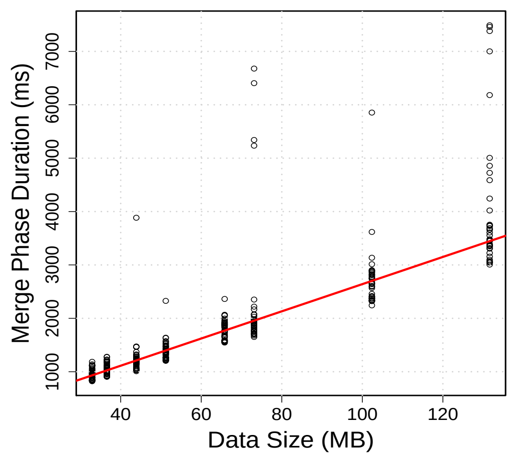
(a) read



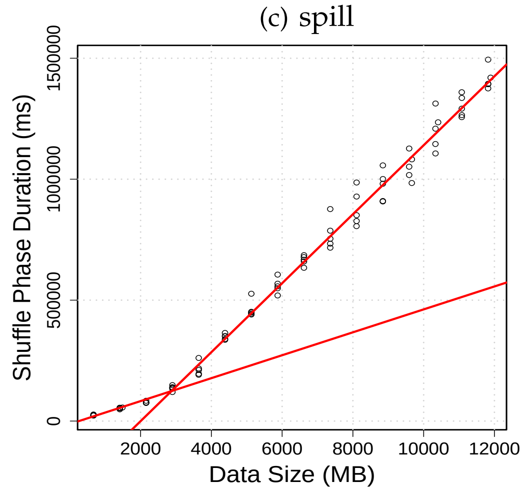
(b) collect



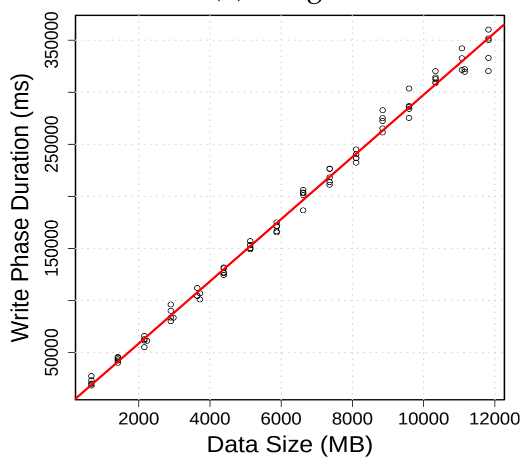
(c) spill



(d) merge



(e) shuffle



(f) write

Figure 3.4: Benchmark results.

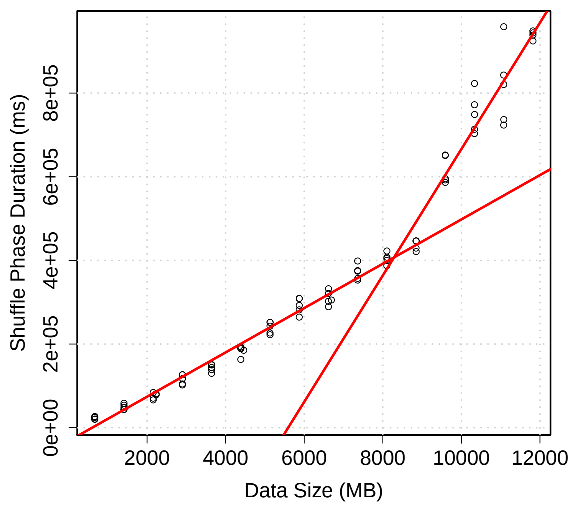


Figure 3.5: Shuffle phase model for Hadoop where each JVM (slot) configured with 2GB of memory.

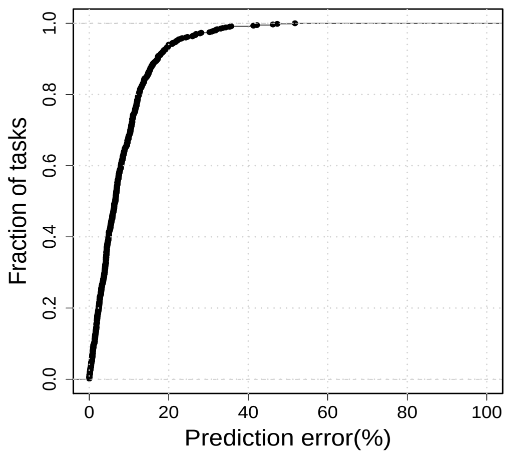
3.1.4 Accuracy of the platform performance model

In order to formally evaluate the accuracy and fit of the generated model M_{Phases} we compute for each data point in our training dataset a prediction error. That is, for each row j in the platform profile we compute the duration dur_i^{pred} of the corresponding phase i by using the derived model M_i as a function of data $Data^j$. Then we compare the predicted value dur_i^{pred} against the measured duration d_i^{measrd}. The relative error is defined as follows:

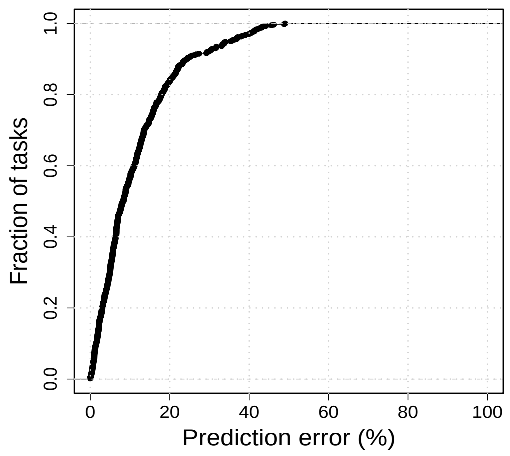
$$error_i = \frac{|d_i^{measrd} - d_i^{pred}|}{d_i^{measrd}}$$

We compute the relative error for all the data points in the platform profile. Figure 3.6 show the CDF of relative errors for all six phases.

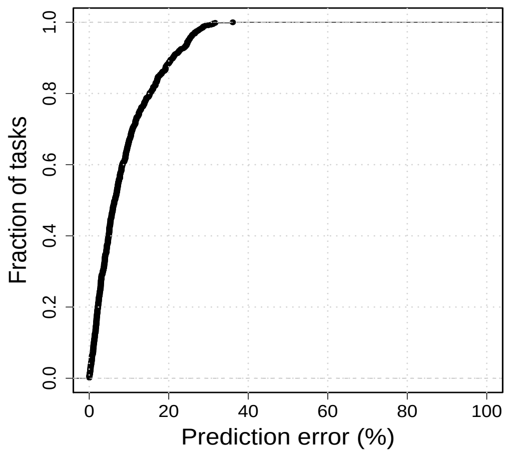
The CDF of relative errors proves that our performance model fits well to the experiment data. Table 3.1 shows the summary of relative errors for derived models of six processing phases. For example, for the *read* phase, 66% of the map tasks have the relative error less than 10% and 92% of the map tasks have the relative error less that 20%. For the *shuffle* phase, 76% of the reduce tasks have the relative error less that 10% and 96% of the reduce tasks have the relative error less that



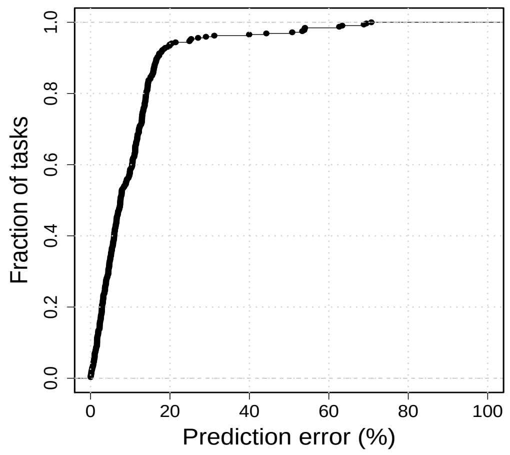
(a) read



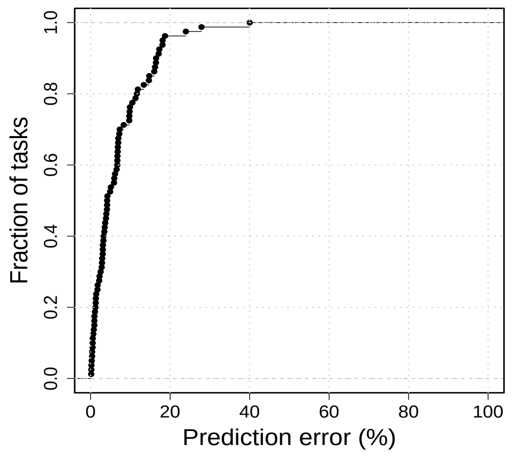
(b) collect



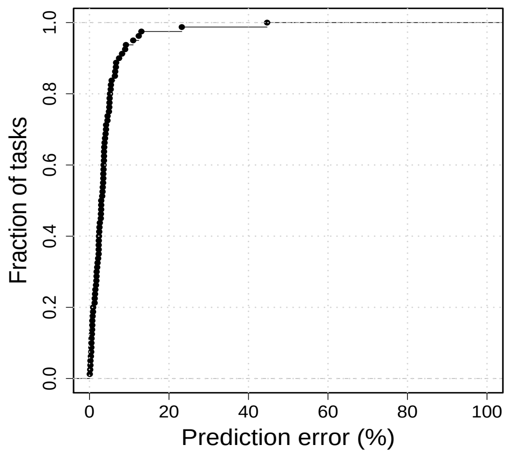
(c) spill



(d) merge



(e) shuffle



(f) write

Figure 3.6: CDF of prediction error.

20%. In summary, almost 80% of all the predicted values are within 15% of the corresponding measurements. Thus the derived platform performance model fits well the collected experimental data.

Table 3.1: Relative error distribution

phase	error $\leq 10\%$	error $\leq 15\%$	error $\leq 20\%$
read	66%	83%	92%
collect	56%	73%	84%
spill	61%	76%	83%
merge	58%	84%	94%
shuffle	76%	85%	96%
write	93%	97%	98%

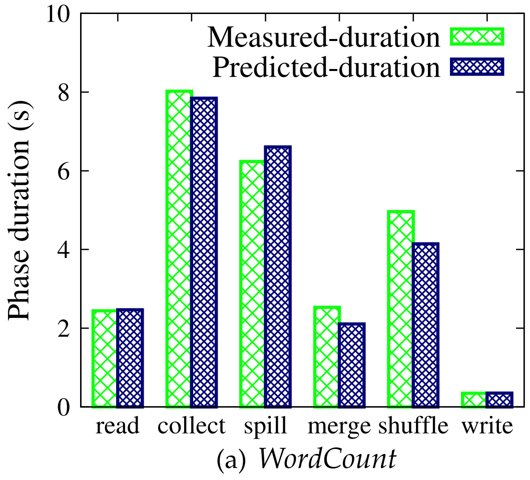
Next, we validate the accuracy of the constructed platform performance model for predicting different phase durations of two example applications provided with Hadoop – *TeraSort* and *WordCount*. We execute these applications on the same 5-node cluster and compare the measured phase durations with the predicted phase completion time based on our model. The input data used by both application is generated using the *TeraGen* program with a total size of 2.5 GB.

Figure 3.10 shows the comparison of the measured [1] and predicted durations for 6 generic execution phases. The number of reduce tasks is fixed in these experiments and set to 40 in both jobs. The graphs reflect that the constructed performance model could accurately predict the durations of each phase as a function of the processed data. The differences between the measured and predicted durations are within 10% in most cases (only for the shuffle phase of *WordCount* application the difference is around 16%).

The next question to answer is whether the model constructed in the small test cluster can be effectively applied for modeling the application performance in the larger production clusters?

To answer this question we execute the same jobs (with the scaled input dataset of 7.5 GB) on the large production cluster in HP Labs which consists of 66 HP

[1] All the experiments are performed five times, and the measurement results are averaged. This comment applies to the results in Figure 3.10 and 3.11.



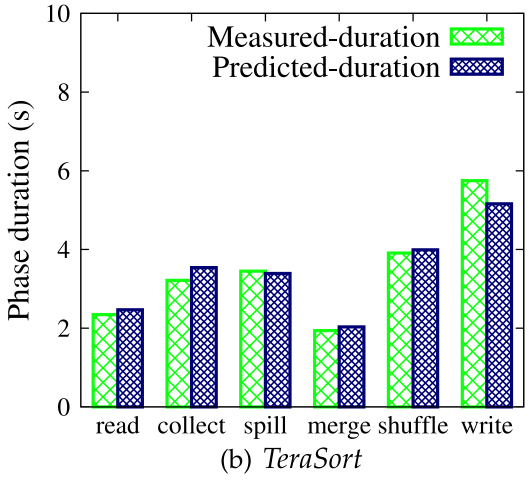
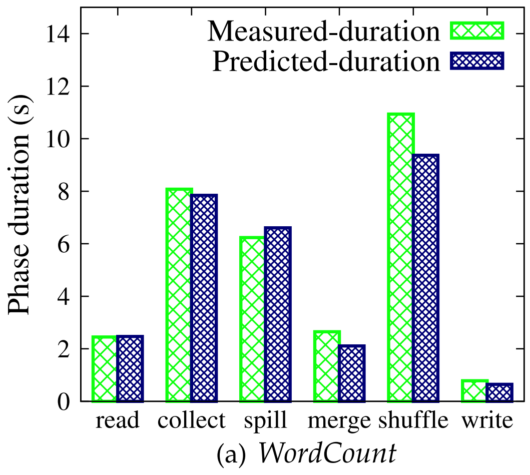


Figure 3.7: Validating the accuracy of the platform performance model on the *small 5-node test cluster*.



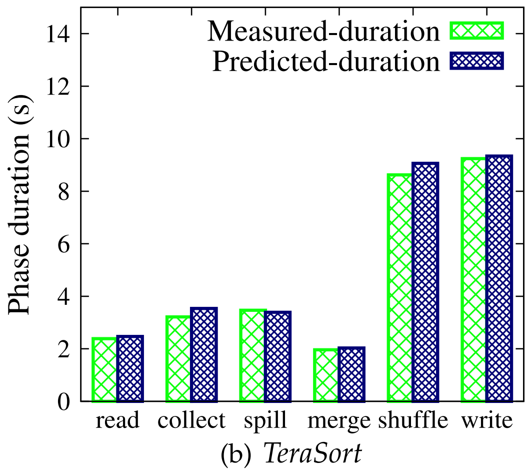


Figure 3.8: Validating the accuracy of platform performance model on the *large 66-node production cluster*.

DL145 GL3 machines. Each machine has the same hardware as the one in our test cluster. The machines are set up in two racks and interconnected with gigabit Ethernet. We used Hadoop 0.20.2 with two machines dedicated as the JobTracker and the NameNode, and remaining 64 machines as workers. Each worker is configured with 2 map and 1 reduce slots. The number of reduce tasks is fixed and set to 60 in both applications.

Figure 3.8 shows measured and predicted durations of six processing phases. The predicted phase durations closely approximate the measured ones. These results justify our approach for building the platform performance model by using a small test cluster.

Running benchmarks on the small cluster significantly simplifies the approach applicability, since these measurements do not interfere with production workloads while the collected platform profile leads to a good quality platform performance model that can be efficiently used for modeling production jobs in the larger enterprise cluster.

3.2 MapReduce job model

The next part of our performance modeling framework is a bounds based MapReduce job model for predicting the performance of a MapReduce application with single job. It combines the platform model we described in Chapter 3.1 in approximating execution times of the generic phases, a compact job profile that represent the characteristics of the user-defined map and reduce functions and a bounds based analytical model proposed in ARIA project [66] that estimates the lower and upper bounds on the job completion time as a function of allocated map and reduce slots. The advantage of our approach includes:

- Non-intrusive: To get the job profile, our approach does not require any modifications or instrumentation or either the application or the underlying Hadoop/Pig execution engines.
- Light-weight: The bounds based analytical performance model relies on the average and maximum duration of the map (reduce) task durations and can estimate the application duration instantly once those information is available.

The overall flow of the computation process is shown in Figure 3.9. We first estimate the average and maximum durations for the map and reduce tasks according to the input data set and the job settings using our platform performance model. We then apply the bounds based analytical model to approximate the entire job completion time.

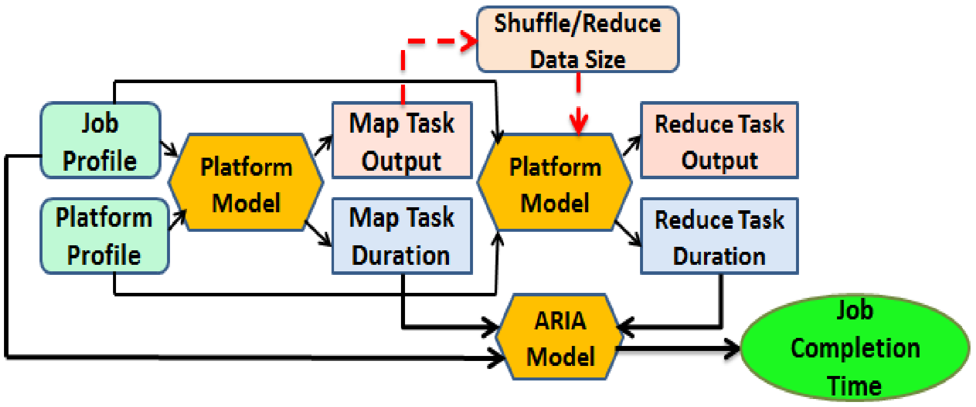


Figure 3.9: MapReduce Performance Model.

3.2.1 Estimate task durations within a job

The first step of our approach is to estimate the map and reduce task durations within a MapReduce job.

Since for each MapReduce job, the map and reduce tasks are consist of a sequence of execution phases, and the the completion time of the tasks can be estimated as the sum of its phase durations. Specifically, for map tasks the completion time is estimated as the sum of the durations of *read*, *map*, *collect*, *spill*, and *merge* phases.

$$T^J_{Mtask} = T^J_{read} + T^J_{map} + T^J_{collect} + T^J_{spill} + T^J_{merge} \tag{3.2}$$

For reduce tasks, the completion time is estimated as the sum of durations for *shuffle*, *reduce*, and *write* phases.

$$T^J_{Rtask} = T^J_{shuffle} + T^J_{reduce} + T^J_{write} \tag{3.3}$$

The phase durations of generic phases (*read*, *collect*, *spill*, *merge*, *shuffle* and *write*) are approximated with the *platform performance model* by applying the derived functions to the data amounts flowing through the phases:

$$T^J_{phase} = M_{phase}(Data^J_{phase}) \tag{3.4}$$

$$phase \in \{read, collect, spill, merge, shuffle, write\}$$

The map (reduce) phase duration depends on the user-defined map (reduce) functions (invoked per record). To model the performance for these customized phases, for a given job J, we extract a special job profile from the previous run of this job. It includes the following metrics:

- the map (reduce) selectivity Sel_M^J (Sel_R^J) that reflects the ratio of the map (reduce) output size to the map (reduce) input size;
- the processing time per record of map (reduce) function $T_{Rec\_map}^J$ ($T_{Rec\_red}^J$).

In addition, we also need to know the number of map and reduce tasks of each job, denoted as N_M^J and N_R^J respectively. Note that N_M^J is determined by the input data of the job, and N_R^J is defined by the job configuration.

With the extracted job profile, the map (reduce) phase duration is directly estimated from the number of input records $Record_{map}^J$ ($Record_{reduce}^J$) and the map (reduce) function processing time per record $T_{Rec\_map}^J$ ($T_{Rec\_red}^J$):

$$T_{map}^J = T_{Rec\_map}^J \times Record_{map}^J \tag{3.5}$$

$$T_{reduce} = T_{Rec\_red}^J \times Record_{reduce}^J \tag{3.6}$$

From the above discussion, we can find that to estimate the phase durations, a critical part of the model is the ability to estimate the data size flow through the phases within a job. We will show next how could we estimate these information given the input dataset.

Given a MapReduce job with certain input dataset(s), we collect the average and maximum data block size (in *bytes* and in the number of *records*) for all the dataset(s). This information determines the average and maximum input sizes per map task in the job, denoted as $Inp_M^{J,avg}$ and $Inp_M^{J,max}$ respectively.

Note, that the input data size for *read* phase equals to the input data size for each map task. The amount of data flowing through *collect*, *spill*, and *merge* phases is estimated by applying the map selectivity Sel_M^J to the input data size (in bytes and in records)[2]. Using the average and maximum input data sizes $Inp_M^{J,avg}$ and

[2]If the combiner is defined for data aggregation and reduction during the spill phase, we apply an additional combiner selectivity $Sel_{M\_comb}^J$, that is measured with special Hadoop counters available for this case.

$Inp_M^{J,max}$, we can estimate the average and maximum map task durations respectively.

The input size for the shuffle phase (i.e., the reduce input size) depends on the map outputs and the number of reduce task number. Let's assume that the map outputs are distributed evenly to each reduce task, than the reduce input size is estimated as

$$Data^J_{shuffle} = (Inp_M^{J,avg} \times Sel_M^J \times N_M^J)/N_R^J \tag{3.7}$$

The input size for the write size is estimated by applying the reduce selectivity to the reduce input size as

$$Data^J_{write} = Data^J_{shuffle} \times Sel_R^J \tag{3.8}$$

3.2.2 Performance model for a single MapReduce job

The proposed performance model for single MapReduce job is based on a general model for computing performance bounds on the completion time of a given set of n tasks that are processed by k servers, (similarly, n map tasks are processed by k map slots in MapReduce environment). Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks in a given set. Let k be the number of servers that can each execute one task at a time. The assignment of tasks to servers is done using an online, *greedy* algorithm: assign each task to the server which finished its running task the earliest. Let avg and max be the *average* and *maximum* duration of the n tasks respectively. Then the completion time of a greedy task assignment is proven to be at least:

$$T^{low} = \frac{n \cdot avg}{k}$$

and at most

$$T^{up} = \frac{(n-1) \cdot avg}{k} + max$$

The difference between lower and upper bounds represents the range of possible completion times due to task scheduling non-determinism. Note, that these provable lower and upper bounds on the completion time can be easily computed if we know the average and maximum durations of the set of tasks and the number of allocated slots.

As motivated by the above model, in order to approximate the overall completion time of a MapReduce job J, we need to estimate the *average* and *maximum* task durations during *map* and *reduce* execution stage of the job. Once we have got (estimated) the average and maximum map (reduce) task durations for a job J (denoted as M_{avg}^J (M_{max}^J) and R_{avg}^J (R_{max}^J)), then, by applying the outlined bounds model and the number of map (reduce) slots available for processing the job, we can estimate the completion times of different processing stage of the job. Note that given a MapReduce job with known input dataset(s), the average and maximum of map (reduce) task duration is independent of the amount of resource assigned to the job, i.e., they represent the *invariant* that characterize the job processing.

For example, let job J be partitioned into N_M^J map tasks. Then the lower and upper bounds on the duration of the entire map stage in the **future** execution with S_M^J map slots (denoted as T_M^{low} and T_M^{up} respectively) are estimated as follows:

$$T_M^{low} = N_M^J \cdot M_{avg}^J / S_M^J \tag{3.9}$$

$$T_M^{up} = (N_M^J - 1) \cdot M_{avg}^J / S_M^J + M_{max}^J \tag{3.10}$$

Similarly, we can compute bounds of the execution time of other processing phases of the job. As a result, we can express the estimates for the entire job completion time (lower bound T_J^{low} and upper bound T_J^{up}) as a function of map (reduce) tasks (N_M^J, N_R^J) and the allocated map (reduce) slots (S_M^J, S_R^J) using the following equation form:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low}. \tag{3.11}$$

where A_J^{low}, B_J^{low} and C_J^{low} represent the coefficient we get from the invariant during the job execution.

The equation for T_J^{up} can be written in a similar form (see [66] for details and exact expressions for the coefficients in these equations). Typically, the average of lower and upper bounds (T_J^{avg}) is a good approximation of the job completion time.

Modeling for heterogeneous clusters

In many practical deployments today, clusters are grown incrementally over time. It is not unusual for companies to start off with an initial cluster, and then gradually add more compute and I/O resources as the number of users increases. More often than not, it is economical to add newer servers with increased compute power to existing clusters, rather than discard the old hardware. As a result, the practical cluster is typically heterogeneous that contains different types of nodes.

We argue that the MapReduce job model also works for heterogeneous environments. Intuitively, in a heterogeneous Hadoop cluster, slower nodes result in longer task executions. These measurements then are reflected in the calculated average and maximum task durations that comprise the job profile. While the bounds-based performance model does not explicitly consider different types of nodes, their performance is implicitly reflected in the job profile and used in the future prediction.

For heterogeneous cluster that consists of groups of nodes of different type, the job profiles can be generated by combining the profiles we get from each type of nodes that the cluster contains. Specifically, the average task durations are generated according to the weighted average of the average durations when executed on each type of nodes and the maximum durations are from the max of the durations from each type.

3.2.3 Accuracy of the MapReduce job model

To validate the accuracy of the MapReduce job model, we use the same hardware platform as we described in Chapter 3.1.4 and use a set of 13 applications made available by the Tarazu project [13]:

1. *Sort* sorts randomly generated 100-byte tuples. The sorting occurs in MapReduce framework's in-built sort while map and reduce are identity functions.
2. *WordCount* counts all the unique words in a set of documents.
3. *Grep* searches for an input string in a set of documents.
4. *InvertedIndex* takes a list of documents as input and generates word-to-document indexing.
5. *RankedInvertedIndex* takes lists of words and their frequencies per file as input, and generates lists of files containing the given words in decreasing order of frequency.
6. *TermVector* determines the most frequent words on a host (above a specified cut-off) to aid analysis of the host's relevance to a search.
7. *SequenceCount* generates a count of all unique sets of three consecutive words per document in the input data.
8. *SelfJoin* is similar to the candidate generation part of the Apriori data mining algorithm. It generates association among k+1 fields given the set of k-field associations and uses synthetically generated data as input.
9. *AdjacencyList* is useful in web indexing to generate adjacency and reverse adjacency lists of nodes of a graph for use by PageRank-like algorithms. It uses synthetically generated web graph based on a Zipfian distribution.
10. *HistogramMovies* generates a histogram of the number of movies with different average ratings (from 1 to 5).
11. *HistogramRatings* generates a histogram of all user ratings (ranging from 1 to 5).
12. *Classification* classifies the input movies into one of k pre-determined clusters using the cosine-vector similarity.
13. *KMeans* clusters movies into k clusters in a similar way as Classification and recomputes the new centroids afterwards.

Table 3.2 provides a high-level summary of these 13 applications with the corresponding job settings (e.g, number of map and reduce tasks). Applications 1, 8, and 9 process synthetically generated data, applications 2 to 7 use the Wikipedia articles dataset as input, while applications 10 to 13 use the Netflix movie ratings dataset. We present results of running these applications with: *i) small input datasets* defined by parameters shown in columns 3-4, and *ii) large input datasets* defined by parameters shown in columns 5-6 respectively.

Table 3.2: Application characteristics.

Application	*Input data (type)*	*Input (GB) small*	*#Map, Reduce tasks*	*Input (GB) large*	*#Map, Reduce tasks*
1. *TeraSort*	Synthetic	2.8	44, 20	31	495, 240
2. *WordCount*	Wikipedia	2.8	44, 20	50	788, 240
3. *Grep*	Wikipedia	2.8	44, 1	50	788, 1
4. *InvIndex*	Wikipedia	2.8	44, 20	50	788, 240
5. *RankInvIndex*	Wikipedia	2.5	40, 20	46	745, 240
6. *TermVector*	Wikipedia	2.8	44, 20	50	788, 240
7. *SeqCount*	Wikipedia	2.8	44, 20	50	788, 240
8. *SelfJoin*	Synthetic	2.1	32, 20	28	448, 240
9. *AdjList*	Synthetic	2.4	44, 20	28	508, 240
10. *HistMovies*	Netflix	3.5	56, 1	27	428, 1
11. *HistRatings*	Netflix	3.5	56, 1	27	428, 1
12. *Classification*	Netflix	3.5	56, 16	27	428, 50
13. *KMeans*	Netflix	3.5	56, 16	27	428, 50

Validation in homogeneous environment

Figure 3.10 shows the comparison of the measured and predicted job completion times [3] for 13 applications (with a small input dataset) executed using 5-node test cluster. The graphs reflect that the designed MapReduce performance model closely predicts the job completion times. The measured and predicted durations are less than 10% for most cases (with 17% error being a worst case for *WordCount* and *HistRatings*). Note the split at Y-axes in order to accommodate a much larger scale for a completion time of the *KMeans* application.

The next question to answer is whether the *platform performance model* constructed using a small 5-node test cluster can be effectively applied for modeling the

[3] All the experiments are performed five times, and the measurement results are averaged. This comment applies to the results in Figure 3.10, 3.11.

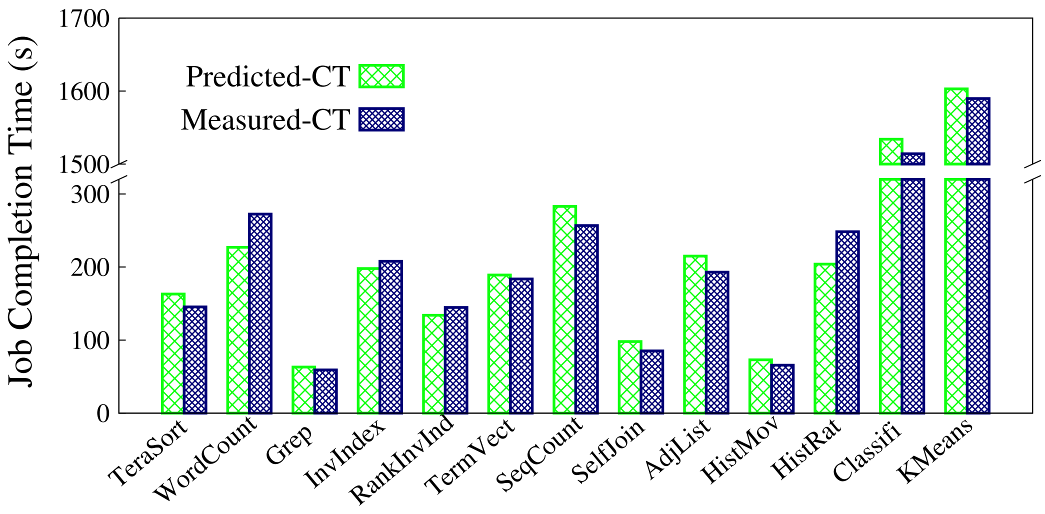


Figure 3.10: Predicted vs. measured completion times of 13 applications on the *small 5-node test cluster*.

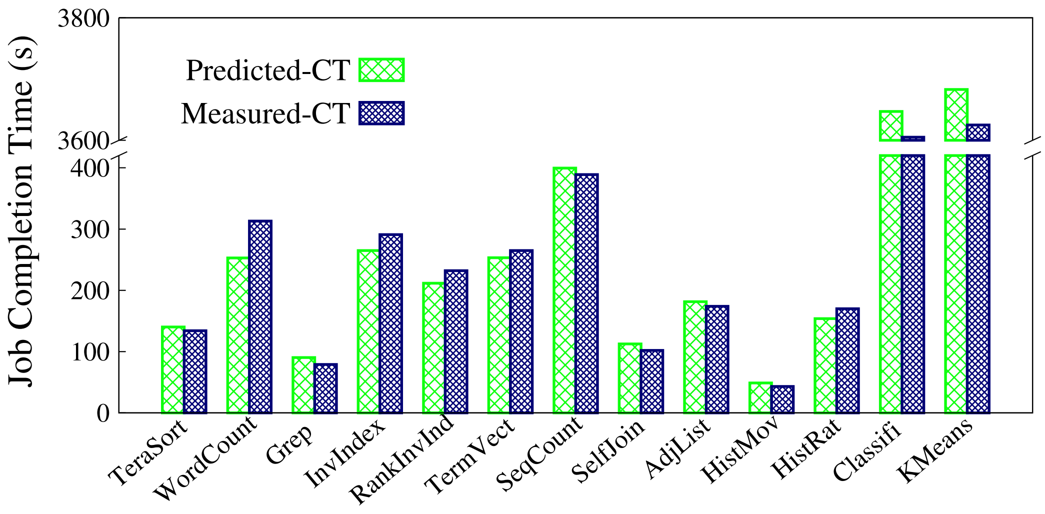


Figure 3.11: Predicted vs. measured completion times of 13 applications (with a large input dataset) on the *large 66-node production cluster*.

application performance in the larger production clusters. To answer this question we execute the same 12 applications (with a large input dataset) on the 66-node production cluster. Figure 3.11 shows the comparison of the measured and predicted job completion times for 13 applications executed on the 66-node Hadoop cluster. The predicted completion times closely approximate the measured ones: for 12 applications they are less than 10% of the measured ones (a worst case is

WordCount that exhibits 17% of error). Note the split at Y-axes for accommodating the *Classification* and *KMeans* completion time in the same figure.

These results justify our approach for building the platform performance model by using a small test cluster. Running benchmarks on the small cluster significantly simplifies the approach applicability, since these measurements do not interfere with production workloads while the collected platform profile leads to a good quality platform performance model that can be efficiently used for modeling production jobs in the larger enterprise cluster.

Validation in heterogeneous environment

We evaluate the accuracy of the bounds-based performance model for predicting the job completion time in heterogeneous environments using the following two platforms in our experiments. The workload we used is the same application set we used in our homogeneous experiments.

The UPenn heterogeneous cluster It contains 36 worker nodes of 3 different types as shown in Table 3.3.

Table 3.3: UPenn cluster description.

Node type	#nodes	CPU type	RAM (GB)	#m,r slots
*Type*1	16	Xeon X3220 (quad-core) compute nodes, Quad Core Intel Xeon X3220, 2.40GHz	4.0	2, 1
*Type*2	12	Xeon X3363 (quad-core) compute nodes, Quad Core Intel Xeon X3363, 2.83GHz	4.0	2, 1
*Type*3	8	Xeon X3450 (quad-core) compute nodes, Quad Core Intel X3450 Xeon 2.66GHz	4.0	2, 1

The heterogeneity is caused by the extension of the cluster over time. In 2007, the cluster had 16 nodes with the same hardware (*Type1* nodes). Then, two years later, 12 more nodes were added to the cluster with more powerful CPUs (*Type2* nodes). Finally, in 2010, 8 more nodes (*Type3*) were added to the cluster to satisfy the growing workloads and computing demands. Each node has a Dual SATA 250GB drive. All the nodes are connected to the same rack: each node has 1 Gbit/s network connection to a 10 Gbit/s switch. An additional server node (*Type3*) runs the NameNode and JobTracker of the deployed Hadoop cluster. While the n-odes in the UPenn cluster represent different server generations, they all have the same number of CPU cores and the same amount of RAM. This explains why we configure these nodes in a similar way, with the same number of map and reduce slots.

The Amazon EC2 platform The EC2 environment offers a choice of different ca-pacity Virtual Machines (VMs) for deployment. These VMs can be deployed on a variety of hardware and be allocated different amounts of system resources. We build a *heterogeneous* Hadoop cluster that consists of different VM types:

- **10** VMs based on *small instances (m1.small)*,
- **10** VMs based on *medium instances (m1.medium)*, and
- **10** VMs based on *large instances (m1.large)*.

The description of each VM instance type is shown in Table 3.4. Since the compute and memory capacity of a medium VM instance is doubled compared to a small VM instance (similarly, large VM instances have a doubled capacity compared to the medium ones), we configured different numbers of map and reduce slots for different VM instances as shown in Table 3.4. Each VM instance is deployed with 100GB of Elastic Block Storage (EBS).

Table 3.4: EC2 Testbed description.

Instance type	#VMs	*CPU capacity (relative)*	RAM (GB)	#m,r slots
Small	10	1 EC2 Compute Unit (1 virtual core with 1 EC2 Compute Unit)	1.7	1, 1
Medium	10	2 EC2 Compute Unit (1 virtual core with 2 EC2 Compute Unit)	3.75	2, 1
Large	10	4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each)	7.5	4, 4

We use an additional *high-CPU VM instance* for running the NameNode and JobTracker of the Hadoop cluster.

Figure 3.12 shows the predicted vs measured results for 13 applications processed on the UPenn heterogeneous Hadoop cluster.[4] Given that the completion times of different programs range between 80 seconds (for *HistMovies*) and 48 minutes (for *KMeans*), we normalize the predicted completion times with respect to the measured ones for the sake of readability and comparison.

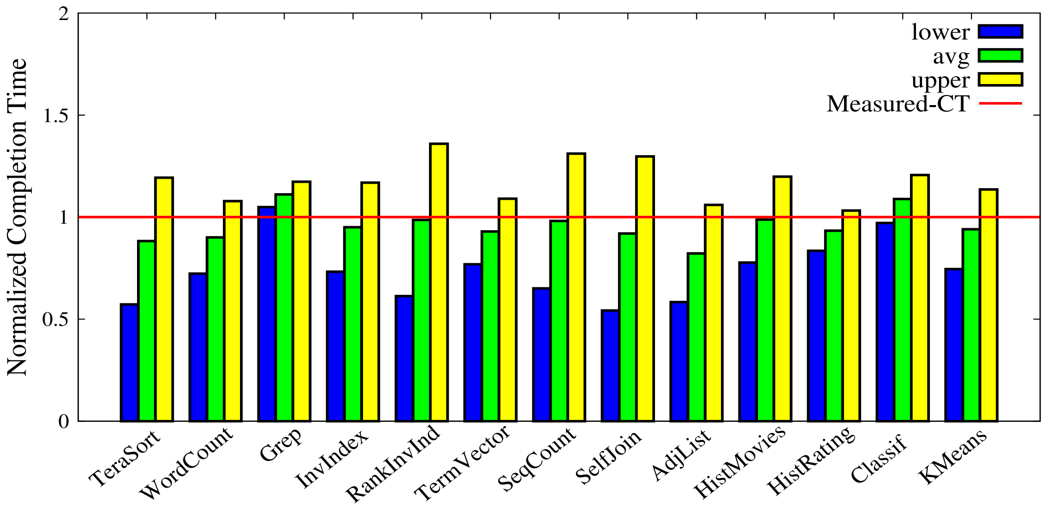


Figure 3.12: Predicting the job completion time in the UPenn cluster.

The three bars in Figure 3.12 represent the normalized predicted completion times based on the lower (T^{low}) and upper (T^{up}) bounds, and the average of them (T^{avg}). We observe that the actual completion times (shown as the straight *Measured-CT* line) of 13 programs fall between the lower and upper bound estimates (except for the *Grep* application). Moreover, the predicted completion times based on the average of the upper and lower bounds are within 10% of the measured results for 11 out of the 13 applications. The worst prediction is around 18% error for the *AdjList* application.

The UPenn cluster contains servers of three different CPU generations, but each node has a similar number of CPU cores, memory, disk storage and network bandwidth. The bounds-based model does work well in this environment. Now, we perform a similar comparison for EC2-based heterogeneous cluster, where the

[4]All the experiments are performed five times, and the measurement results are averaged. This comment also applies to the results in Figure 3.13.

cluster nodes are formed by VM instances with very different computing capacities.

Figure 3.13 shows the normalized predicted completion times (based on the lower, upper, and average bounds) compared to the measured ones for executing 13 applications on the EC2-based heterogeneous cluster. The results validate the accuracy of the proposed model: the measured completion times of all 13 programs fall between the lower and upper bound estimates. The average of the lower and upper bounds are within 10% of the measured value for 9 out of 13 applications with a worst case of 13% error for the *WordCount* application.

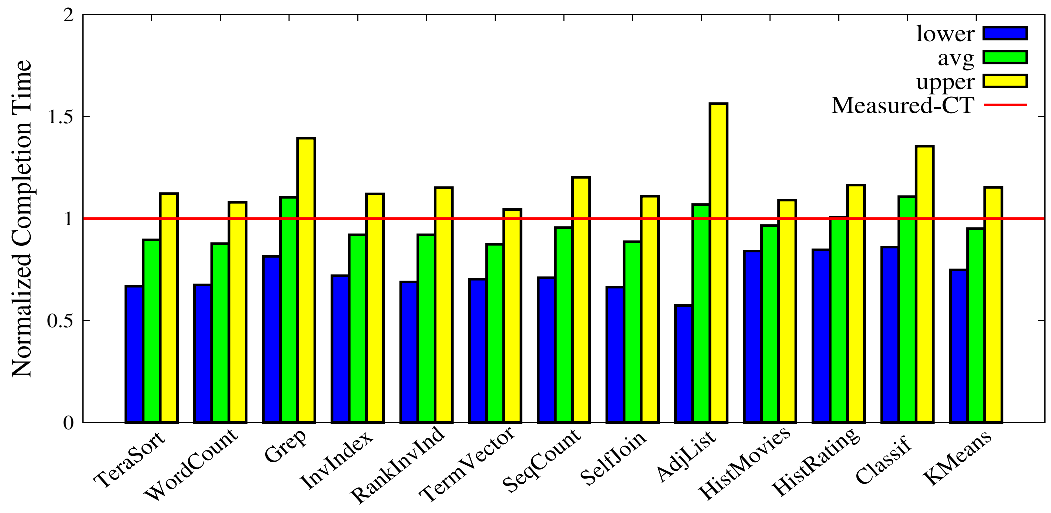


Figure 3.13: Predicting job completion time in heterogeneous EC2 cluster.

For a further analysis on validating the accuracy of job profiles we generated for heterogeneous cluster, we use a sub-cluster that consists of two different kind of node: 8 nodes of *Type1* and 8 nodes of *Type2*. Table 3.5 and Table 3.6 shows the job profiles for two applications: *Adjlist* and *WordCount* when they are executed on homogeneous cluster based on nodes of Type1(Type2) as well as the heterogeneous cluster. The results show that the cluster of node Type2 has a shorter average and maximum task durations as nodes of Type2 have more powerful processors. The average map and reduce task duration when executed on the heterogeneous cluster are in between of the average durations when they are executed on the other two homogeneous clusters while the maximum task durations are very close to the max of task durations extracted from the execution on homogeneous clusters.

	avg map	max map	avg reduce	max reduce
Type1	106s	132s	723s	764s
Type2	94s	117s	634s	654s
Heterogeneous	100s	131s	679s	760s

Table 3.5: Job profiles of *Adjlist* on UPenn cluster.

	avg map	max map	avg reduce	max reduce
Type1	19s	28s	44s	58s
Type2	14s	22s	38s	46s
Heterogeneous	17s	27s	43s	60s

Table 3.6: Job profiles of *WordCount* on UPenn cluster.

Among the 13 applications, there are 3 special ones: *Grep*, *HistMovies*, and *HistRatings*, which have a single reduce task (defined by the special semantics of these applications). The shuffle/reduce stage durations of such an application depend on the Hadoop node type that is allocated to execute this single reduce task. If there is a significant difference in the Hadoop nodes, it may impact the completion times of the shuffle/reduce stages across the different runs, and therefore, make the prediction inaccurate.

Figure 3.14 analyzes the executions of *Grep*, *HistMovies*, and *HistRatings*, on the heterogeneous EC2-based cluster.

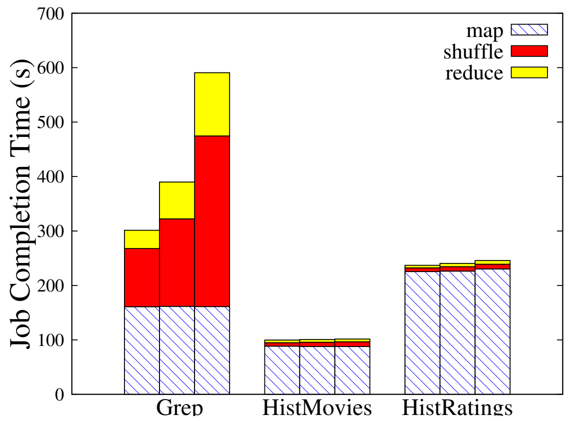


Figure 3.14: A special case of jobs with a single reduce task: their possible executions on the heterogeneous EC2 cluster.

The three bars within each group represent (from left to right) the job completion times when the reduce task is executed by the reduce slot of a *large*, *medium*, or *small* VM instance. Each bar is further split to represent the relative durations of the map, shuffle, and reduce stages in the job execution. We observe that the completion time of the *Grep* application is significantly impacted by the type of VM instance allocated to the reduce task execution. The reduce task execution time on the *small* VM instance is almost twice as long as that on a large VM instance

In comparison, the execution times of *HistMovies* and *HistRatings* on different capacity VMs are not significantly different, because for these applications the reduce task duration constitutes a very small fraction of the overall completion time (see Figure 3.14). In summary, if the execution time of a reduce stage constitutes a significant part of the total completion time, the accuracy of the bounds-based model may be adversely impacted by the node type allocated to the reduce task execution.

3.3 MapReduce workflow peformance model

Next, we further explain our performance model in predicting the completion time for more complex MapReduce applications that contain a DAG of MapReduce jobs, i.e., MapReduce workflows. It is based on the MapReduce job model we explained in Chapter 3.2. We first introduce a single basic approach in Chapter 3.3.2, We show that such basic approach is efficient for estimating completion time for a workflow consists of sequential MapReduce jobs, but is pessimistic for workflows that contains concurrent jobs. We then propose a refined approach to address workflows with concurrent execution in Chapter 3.3.3. We validate the accuracy of our approach with several different workloads.

3.3.1 Estimate input data size through the worklfow

Before we are going to describe our approach in estimating the workflow durations, we need to first explain the method we used to estimate the amount of data flow through each job within the workflow. These information is then used to estimate the duration of each phase in the MapReduce pipeline by using the platform mode we introduced in Chapter 3.1 that forms the basis of our evaluation framework.

Among the input datasets of a MapReduce jobs in a workflow, we distinguish *external* and *internal* datasets. The external datasets reside in HDFS and exist prior to the workflow execution. For example, the first job in a workflow has only external input datasets. The input of a intermediate job in a workflow is defined by the output of previous job. We call such input datasets as internal ones.

For an intermediate job in a given workflow, the input data size per map task depends on the following factors:

- the output size of the previous job,
- the number of reduce tasks of the previous job, and
- the block size on HDFS.

In particular, each reduce task generates an output file which is stored in HDFS. If the output file size is larger than the HDFS block size (default value 64MB), the output file will be split into multiple data blocks, and each of them will be read by a map task of the next job. For example, let the output size be 70 MB. In this case, this output will be written as two blocks: one of 64 MB and the second of 6 MB, and it will define two map tasks that read files of varying sizes (64 MB and 6 MB). Based on these observations, we can estimate the number of map tasks and the average map input size of the next jobs as

$$N_M^{J_i} = \lceil Data_{write}^{J_{i-1}} / Data_{block} \rceil \times N_R^{J_{i-1}} \tag{3.12}$$

$$Inp_M^{J_i} = (Data_{write} \times N_R) / N_M^{J_i} \tag{3.13}$$

For jobs that read from multiple datasets (e.g, jobs that perform the join operation), we get the job profiles and the input data information for each dataset and estimate the average and maximum map task durations based on these information (denoted as $T_{M_I}^{J,avg}$ and $T_{M_I}^{J,max}$ respectively. Specifically, suppose given a job J

with K different input datasets, we have

$$T^{J,avg}_{M\_task} = \frac{\sum_{1\leq i\leq K} T^{J,avg}_{M_I} \times N^J_{M_I}}{\sum_{1\leq i\leq K} N^J_{M_i}} \quad (3.14)$$

$$T^{J,max}_{M\_task} = max_{1\leq i\leq K} T^{J,max}_{M_i} \quad (3.15)$$

3.3.2 Modeling MapReduce workflows with sequential jobs

Consider a MapReduce workflow W that contains N MapReduce jobs $W = \{J_1, J_2, ...J_N\}$. We show first a simple approach to estimate the workflow completion time according to the input dataset and allocated map and reduce slots for the workflow (denoted as S^W_M and S^W_R respectively). It is based on the platform performance model and the MapReduce job model we introduced in Chapter 3.1 and Chapter 3.2 respectively.

According to the description in Chapter 3.2, for a given MapReduce workflow with N jobs, we can estimate the average and maximum map and reduce task durations for each job J_i within the workflow given the workflow input datasets. Then, by applying the bounds based model outlined in Chapter 3.2.2 and the estimated average (maximum) task durations, we are able to approximate the lower and upper bound of completion time of each job J_i that belongs to the workflow as a function of S^W_M and S^W_R in the following form:

$$T^{low}_{J_i}(S^W_M, S^W_R) = A^{low}_{J_i} \cdot \frac{N^{J_i}_M}{S^W_M} + B^{low}_{J_i} \cdot \frac{N^{J_i}_R}{S^W_R} + C^{low}_{J_i} \quad (3.16)$$

For a workflow that contains only sequential jobs, a straightforward approach is to estimate the overall program completion time as a sum of completion times of all the jobs that constitute the workflow:

$$T^{low}_W(S^W_M, S^W_R) = \sum_{1\leq i\leq N} T^{low}_{J_i}(S^W_M, S^W_R) \quad (3.17)$$

The computation of the estimates based on different bounds (T^{up}_W and T^{avg}_W) are handled similarly: we use the respective models for computing T^{up}_J or T^{avg}_J for each MapReduce job J_i $(1 \leq i \leq N)$ that constitutes the workflow.

If individual MapReduce jobs within the workflow are assigned different number of slots, our approach is still applicable: we would need to compute the completion time estimates of individual jobs as a function of their individually assigned resources.

Evaluate the effectiveness of the workflow performance model

We evaluate the accuracy of the proposed performance model using the same testbed as we described in Chapter 3.1.4.

We use the well-known **PigMix** [5] benchmark as our case for study. The benchmark was created for testing Pig system performance. It consists of 17 Pig programs (L1-L17), which uses datasets generated by the default Pigmix data generator with 8 tables. The details about the table layout and the query set can be found in [5]. In our experiments, we generate 125 million records for the largest table and has a total size around 1 TB across 8 tables. The PigMix programs cover a wide range of the Pig features and operators, and the data set are generated with similar properties to Yahoo's datasets that are commonly processed using Pig. With the exception of L11 (that contains a stage with 2 concurrent jobs), all PigMix programs involve DAGs of sequential jobs.

We first run the benchmark to build the specific job profiles with the map (reduce) selectivity and the execution time per record for the map(reduce) function. By using the extracted job profiles and the designed workflow performance model described above, we compute the completion time estimates of Pig programs in the benchmarks as a function of allocated resources. Then we validate the predicted completion times against the measured ones. We execute each benchmark three times and report the measured completion time averaged across 3 runs.

Figure 3.15 shows the results for the PigMix benchmark when each program in the set is processed with 128 map and 64 reduce slots. Given that the completion times of different programs in PigMix are in a broad range of 100s – 2000s, for presentation purposes and easier comparison, we **normalize** the predicted completion times with respect to the measured ones. The three bars in Figure 3.15 rep-

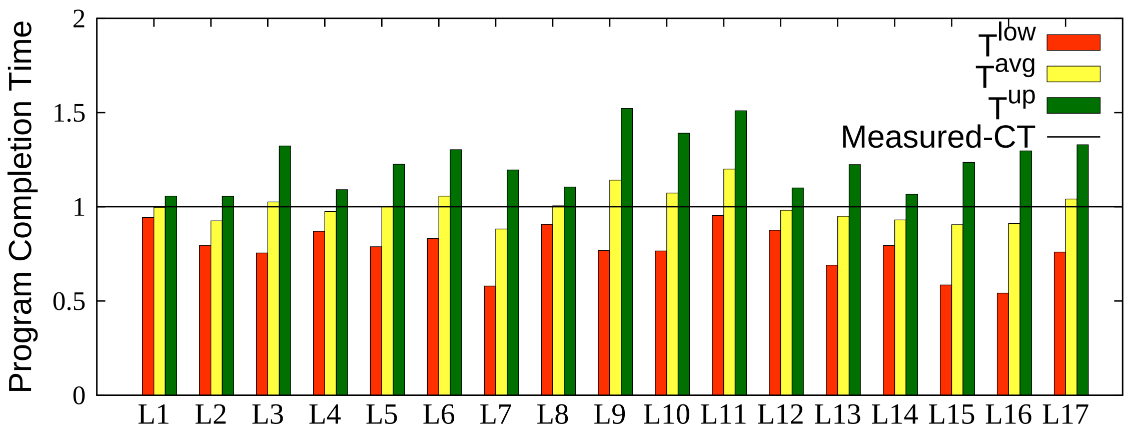


Figure 3.15: Predicted and measured completion time for PigMix with 128x64 slots.

resent the predicted completion times based on the lower (T^{low}) and upper (T^{up}) bounds, and the average of them (T^{avg}). We observe that the actual completion times (shown as the straight *Measured-CT* line) of all 17 programs fall between the lower and upper bound estimates. Moreover, the predicted completion times based on the average of the upper and lower bounds are within 10% of the measured results for most cases. The worst prediction (around 20% error) is for the Pig query L11. The measured completion time of L11 is very close to the lower bound. Note, that the L11 program is the only program in PigMix that is defined by a DAG with concurrent jobs.

Figure 3.16 shows the results for the PigMix benchmark when each program in the set is processed with 64 map and 64 reduce slots. Indeed, our model accurate computes the program completion time estimates as a function of allocated resources: the actual completion times of all 17 programs are in between the computed lower and upper bounds. The predicted completion times based on the average of the upper and lower bounds provide the best results: 10-12% of the measured results for most cases.

Limitation of the performance model

The proposed performance model works well for the PigMix benchmark, however, as most the Pig programs within PigMix is compiled into MapReduce workflows with sequential jobs. It is not clear about the effectiveness of the model in predict-

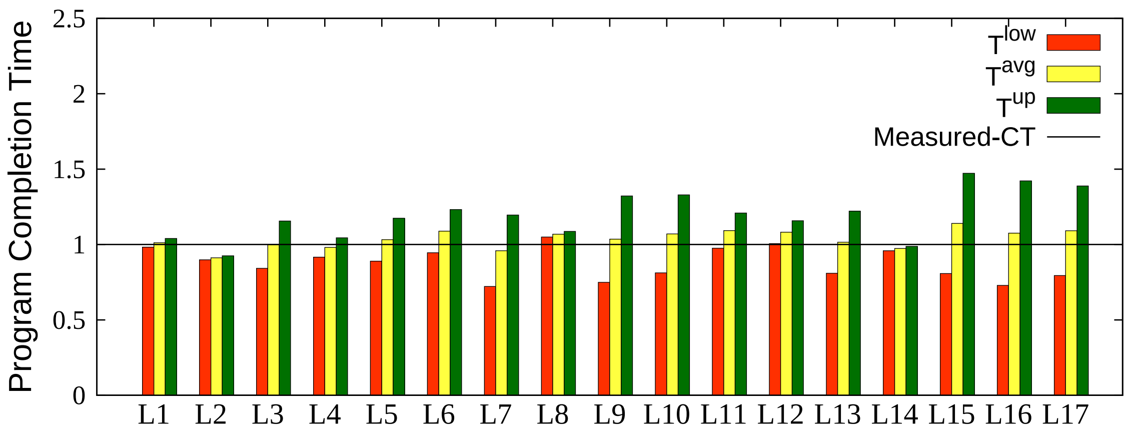


Figure 3.16: Predicted and measured completion time for PigMix with 64x64 slots.

ing the completion time for more general workflows with both sequential and concurrent jobs. To get a better understanding about the accuracy of the performance model for general MapReduce workflow, we performed similar experiments for two other workloads. One of the workloads is based on TPC-H queries, and the other consists of customized queries for mining web proxy logs from HP Labs. We briefly describe the datasets and queries we used in these two workflow as follows:

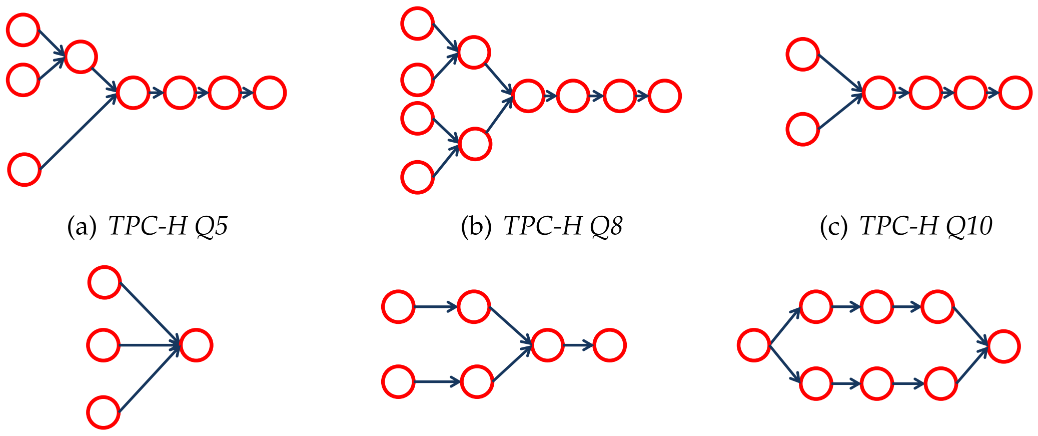


Figure 3.17: DAGs of Pig programs in the TPC-H and HP Labs Proxy query sets.

TPC-H. This workload is based on TPC-H [12], a standard database benchmark for decision-support workloads. The TPC-H benchmark comes with a data generator that is used to generate the test database for queries included in the TPC-H suite. There are eight tables: *customer, supplier, orders, lineitem, part, partsupp, nation*, and *region* used by queries in TPC-H. The input dataset size is controlled by the s-caling factor (a parameter in the data generator). The scaling factor of 1 generates 1 GB input dataset. The created data is stored in ASCII files where each file contains pipe-delimited load data for the tables defined in the TPC-H database schemas. The Pig system is designed to process flexible plain text and can load these data easily with its default storage function. We select 3 queries $Q5, Q8, Q10$ out of 22 SQL queries from the TPC-H benchmark and express them as Pig programs. We pick these queries as they result in DAGs with of concurrent MapReduce jobs [5].

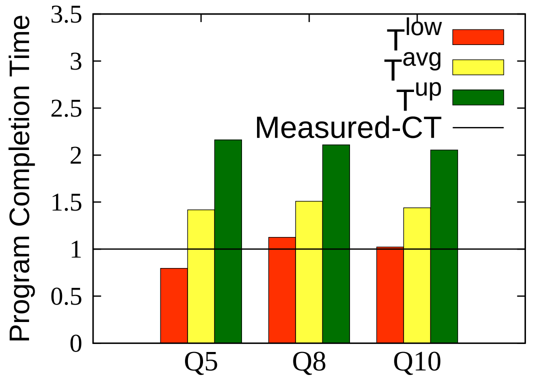
- The *TPC-H Q5* query lists for each nation in a region, the revenue that result-ed from lineitem transactions in which the customer ordering parts and the supplier filling them were both within that nation. It joins 6 tables, and its dataflow results in 3 concurrent MapReduce jobs. The DAG of the program is shown in Figure 3.17 (a).
- The *TPC-H Q8* query determines how the market share of a given nation within a given region has changed over two years for a given part type. It joins 8 tables, and its dataflow results in two stages with 4 and 2 concurrent MapReduce jobs respectively. The DAG of the program is shown in Figure 3.17 (b).
- The *TPC-H Q10* query identifies the customers, who have returned parts that effect on lost revenue for a given quarter, It joins 4 tables, and its dataflow results in 2 concurrent MapReduce jobs with the DAG of the program shown in Figure 3.17 (c).

HP Labs' Web Proxy Query Set. This workload consists of a set of Pig programs for analyzing HP Labs' web proxy logs. The dataset contains 6 months access logs to web proxy gateway at HP Labs during 2011-2012 years. The total dataset size (12 months) is about 36 GB. There are 438 million records in these logs, The proxy

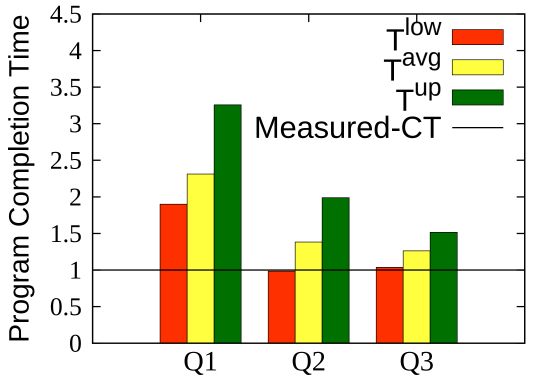
[5]While more efficient logical plans may exist, our goal here is to create a DAG with concurrent jobs to stress test our model.

log data contains one record per each web access. The fields include information such as *date, time, time-taken, c-ip, cs-host*, etc. The log files are stored as plain text and the fields are separated with spaces. Our main intent is to evaluate our models using realistic Pig queries executed on real-world data. We aim to create a diverse set of Pig programs with dataflows that result in the DAGs of MapReduce jobs with concurrent jobs:

- The *Proxy Q1* program investigates the dynamics in access frequencies to different websites per month and compares them across the 6 months. The Pig program results in 6 concurrent MapReduce jobs with the DAG of the program shown in Figure 3.17 (d).
- The *Proxy Q2* program tries to discover the co-relationship between two websites from different sets (tables) of popular websites: the first set is created to represent the top 500 popular websites accessed by web users within the enterprise. The second set contains the top 100 popular websites in US according to Alexa's statistics [6]. The DAG of the Pig program is shown in Figure 3.17 (e).
- The *Proxy Q3* program presents the intersect of 100 most popular websites (i.e., websites with highest access frequencies) accessed both during work and after work hours. The DAG of the program is shown in Figure 3.17 (f).



(a) TPC-H



(b) Proxy Queries

Figure 3.18: Predicted and measured completion time for TPC-H and Proxy queries executed with 128x64 slots.

[6]http://www.alexa.com/topsites

Figure 3.18 shows the normalized results of predicted completion times in respect the actual measured ones for TPC-H and Proxy queries processed with 128 map and 64 reduce slots. As we can find from the figure, the proposed performance model is pessimistic in estimating the completion time for workflows with concurrent jobs – for queries TPC-H Q5, Q10 and Proxy Q2, Q3 the measured completion times are closer to lower bound estimates, and for queries TPC-H Q8 and Proxy Q1, even the lower bound on the predicted completion time is higher than the measured program completion time in the cluster.

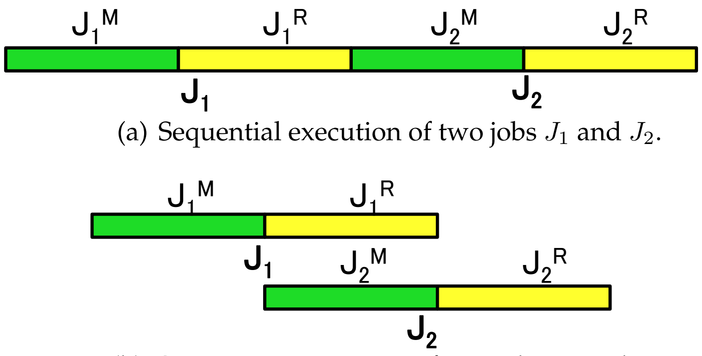
3.3.3 Modeling MapReduce workflows with concurrent jobs

As we shown in Chapter 3.3.2, the simple performance model works well for workflows with sequential jobs. However, it over-estimates the completion time for workflows with concurrent jobs. To understand the reason for such results, we show in this chapter an analysis of the execution of concurrent jobs in the workflow and identify that the execution overlap among the concurrent jobs reduce the overall workflow completion time. Based on the observation, we refine the previous model by incorporating the execution overlap into the model.

Modeling concurrent jobs' executions

Let us consider two concurrent MapReduce jobs J_1 and J_2. There are no data dependencies among the concurrent jobs. Therefore, unlike the execution of sequential jobs where the next job can only start after the previous one is entirely finished (shown in Figure 3.19 (a)), for concurrent jobs, once the previous job completes its map phase and begins reduce phase processing, the next job can start its map phase execution with the released map resources in a pipelined fashion (shown in Figure 3.19 (b)). As a result, with such "overlap" in executions of concurrent jobs, the entire workflow completion time is reduced.

The performance model we proposed in Chapter 3.3.2 approximates the completion time of a workflow $W = \{J_1, J_2\}$ as a sum of completion times of J_1 and J_2



(a) Sequential execution of two jobs J_1 and J_2.

(b) Concurrent execution of two jobs J_1 and J_2.

Figure 3.19: Difference in executions of (a) two sequential MapReduce jobs; (b) two concurrent MapReduce jobs.

(see eq. 3.17) *independent on whether jobs J_1 and J_2 are sequential or concurrent*. While such an approach results in straightforward computations, at the same time, if we do not consider possible overlap in execution of map and reduce stages of concurrent jobs then the computed estimates are pessimistic and over-estimate their completion time.

Modeling MapReduce workflows with concurrent jobs

Now, we explain how we refine the previous performance model for predicting the completion time T_W of a MapReduce workflow W as a function of allocated resources (S_M^W, S_R^W).

Given a MapReduce workflow W that is compiled from a Pig program as a DAG of MapReduce jobs. This DAG represents the Pig program execution plan. The Pig engine partitions the DAG into multiple stages, each stage contains one or more independent MapReduce jobs which can be executed concurrently. For plan execution, the Pig system will first submit all the jobs from the first stage. Once they are completed, it will submit jobs from the second stage, etc. This process continues until all the jobs are completed.

Note that due to the data dependencies within a Pig execution plan, the *next stage* can not start until the *previous stage* finishes. Thus, the completion time of such a MapReduce workflow W which contains S *stages* can be estimated as follows:

$$T_W = \sum_{1 \le i \le S} T_{S_i} \tag{3.18}$$

where T_{S_i} represents the completion time of stage i.

For a stage that consists of a single job J, the stage completion time is defined by the job J's completion time. For a stage that contains concurrent jobs, the stage completion time depends on the completion time of the consisting jobs as well as their execution order.

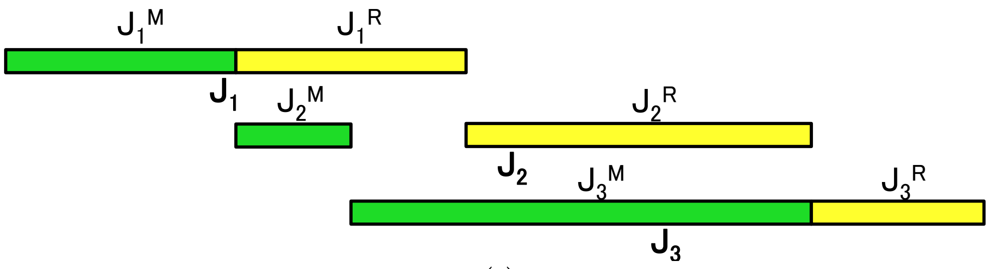
Suppose there are $|S_i|$ jobs within a particular stage S_i and the jobs are executed according to the order $\{J_1, J_2, ...J_{|S_i|}\}$. To describe the model, we use the following notations:

$timeStart^M_{J_i}$	the start time of job J_i's map phase
$timeEnd^M_{J_i}$	the end time of job J_i's map phase
$timeStart^R_{J_i}$	the start time of job J_i's reduce phase
$timeEnd^R_{J_i}$	the end time of job J_i's reduce phase

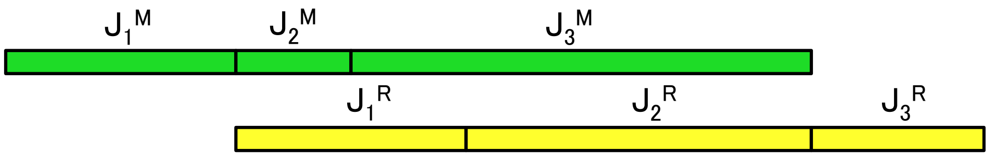
Figure 3.20(a) shows an example of three concurrent jobs execution in the order J_1, J_2, J_3. With the execution overlaps, instead of using the sum of the completion time for each of the consisting job, the stage completion time should be estimated as the time elapses from the start point of the first scheduled job to the end point of the last scheduled job as:

$$T_{S_i} = timeEnd^R_{J_{|S_i|}} - timeStart^M_{J_1} \tag{3.19}$$

We next explain how to estimate the start (end) time of each job's map (reduce) phase. Given the input dataset(s) and the number of allocated map (reduce) slots (S^W_M, S^W_R) to the MapReduce workflow W, we can compute for any MapReduce job $J_i(1 \le i \le |S_i|$ the duration of its map and reduce stages (denoted as $T^M_{J_i}$ and $T^R_{J_i}$ respectively) using the platform model and bounds based estimates as described



(a)



(b)

Figure 3.20: Execution of Concurrent Jobs

in Chapter 3.2.2. [7] Then we have

$$timeEnd_{J_i}^M = timeStart_{J_i}^M + T_{J_i}^M \tag{3.20}$$

$$timeEnd_{J_i}^R = timeStart_{J_i}^R + T_{J_i}^R \tag{3.21}$$

Note, that Figure 3.20 (a) can be rearranged to show the execution of jobs' map (reduce) stages separately as shown in Figure 3.20 (b). It is easy to see that since all the concurrent jobs are independent, the map phase of the next job can start immediately ones the previous job's map stage is finished, i.e.,

$$timeStart_{J_i}^M = timeEnd_{J_{i-1}}^M = timeStart_{J_{i-1}}^M + T_{J_{i-1}}^M \tag{3.22}$$

On the other hand, the start time $timeStart_{J_i}^R$ of the reduce stage of the concurrent job J_i should satisfy the following two conditions:

1. $timeStart_{J_i}^R \geq timeEnd_{J_i}^M$
2. $timeStart_{J_i}^R \geq timeEnd_{J_{i-1}}^R$

Therefore, the start time of the reduce stage can be expressed using the follow-

[7]Here, we use the completion time estimates based on the average of the lower and upper bounds.

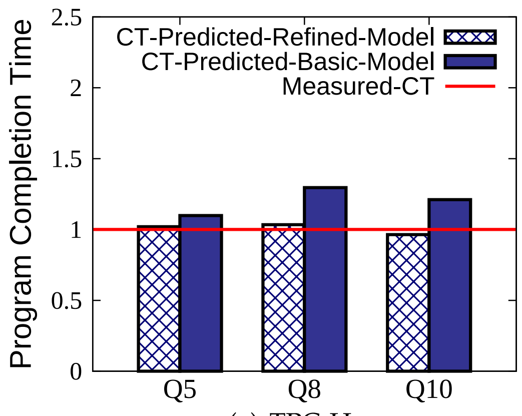
ing equation:

$$timeStart_{J_i}^{R} = max\{timeEnd_{J_i}^{M}, timeEnd_{J_{i-1}}^{R}\} =$$
$$= max\{timeStart_{J_i}^{M} + T_{J_i}^{M}, timeStart_{J_{i-1}}^{R} + T_{J_{i-1}}^{R}\} \quad (3.23)$$

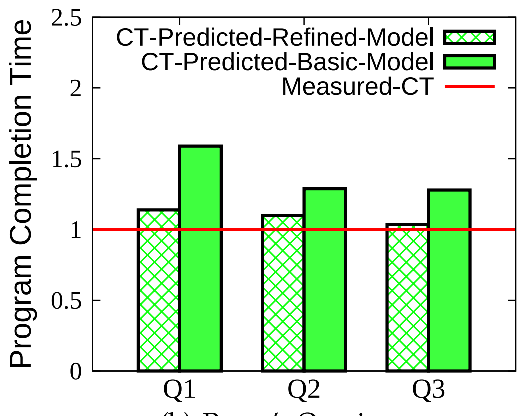
Then the completion time of the entire workflow W is defined as the *sum of its stages* using eq. (3.18).

Evaluate the refined performance model

We evaluate the accuracy of the refined performance model in predicting the completion time for MapReduce jobs with concurrent jobs using the same TPC-H and the HP Lab's proxy query set we used in Chapter 3.3.2. The testbed we used is the same as we described in Chapter 3.1.4. We no longer use the PigMix benchmark because it contains mostly workflows with sequential jobs.



(a) TPC-H



(b) Proxy's Queries

Figure 3.21: Predicted completion times using *basic* vs *refined* models (128x64 slots).

Figure 3.21 shows the workflow completion time estimates based on the performance model introduced in Chapter 3.3.2 (called the *basic* model here) and the *refined* performance model for TPC-H and Proxy queries that represent workflows with concurrent jobs. The completion time estimates are computed using T_W^{avg} (the average of the lower and upper bounds). Figures 3.21 and 3.22 show the results for the case when each program is processed with **128x64** and **32x64** map and reduce slots respectively.

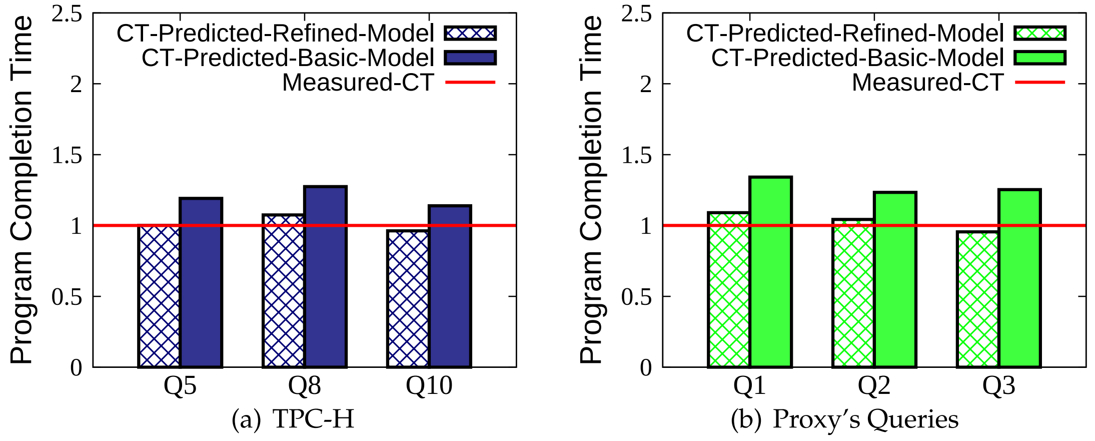


Figure 3.22: Predicted completion times using *basic* vs *refined* models (32x64 slots).

In all cases, the completion time estimates based on the *refined* model are significantly improved compared to the *basic* model results which are too pessimistic. In most cases (11 out of 12), the predicted completion time is within 10% of the measured ones.

Compare with prediction based on linear extrapolation

As a comparison, we also show in the next experiments the prediction results for the TPC-H and proxy queries using a simple linear extrapolation. The experiments are performed on the 64 nodes Hadoop cluster with 2 map slots and 1 reduce slot configured for each node. For TPC-H, we first execute the queries on the data generated with scaling factor equals to 5 and 10 and use the measured durations to derive the linear function to predict the completion time when executed them on a larger dataset generated with scaling factor equals to 20. For proxy queries, we first execute them with the logs collected for 1 and 3 months to derive the linear function and then use the derived functions to predict the performance when executing with the 6-months logs.

The results from Figure 3.23 clearly shows that the linear extrapolation approach does not work well in predicting completion time for MapReduce worklfows. In most cases (5 out 6), the completion time estimates have a difference more that 30% from the measured results with a worst case of 90%. The inaccuracy is caused by 1) the linear extrapolation does not capture the different impact of input data set on different phases and 2) the linear extrapolation does not handle the execution overlap among concurrent executions.

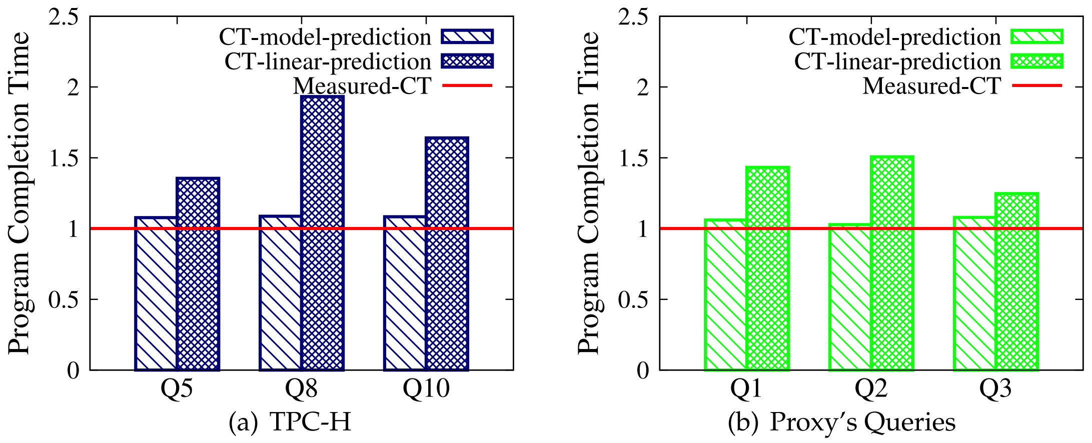


Figure 3.23: Predicted completion time using linear extrapolation (128x64 slots).

3.4 Model sensitivity

In the following part, we discuss a set of factors that could affect the accuracy of our modeling framework which include: 1) impact of sample data size where we use to extract the job profile (Chapter 3.4.1). 2) impact of the characteristics of input data on the map function performance (Chapter 3.4.2). 3) impact of data skew in the reduce stage (Chapter 3.4.3). We also investigate the stability of the job execution across different runs (Chapter 3.4.4). Through the discussion, we try to identify the limitations of our approach and also the applicable applications for our framework.

3.4.1 Impact of sample data size

In designing our modeling framework, we exploit the fact that a typical production MapReduce application is executed routinely on new data. We take advantage of this observation, and for a periodic application, we automatically build its jobs' profiles from the past execution. These extracted job profiles are used for **future** predictions when this application is executed on new data. The question is how sensitive the model and the prediction results to a given training data (i.e. to the extracted job profile from the last job execution).

In our profiling approach, we estimate processing costs per record during the map and reduce phases. This normalized cost is used to project the map/reduce phase duration when these tasks need to process the increased (or decreased) amounts of data (records). Figure 3.24 shows a predicted completion time (normalized with respect to the measured time) of *TPC-H Q10* with different scale factors. Our prediction is based on the job profiles extracted from *TPC-H Q10* with the scale factor 9.

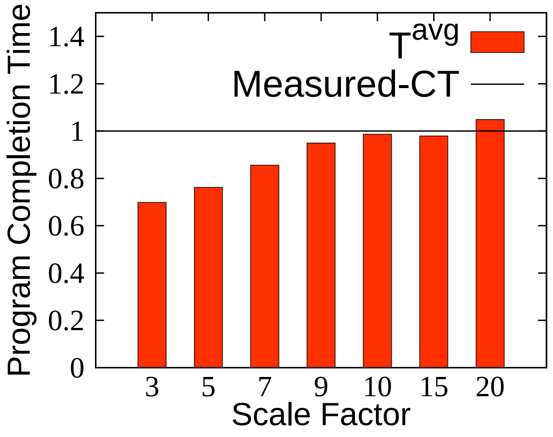
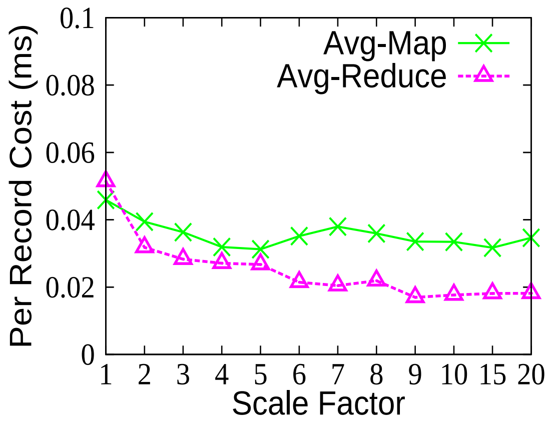


Figure 3.24: Impact of sample data size on completion time estimates

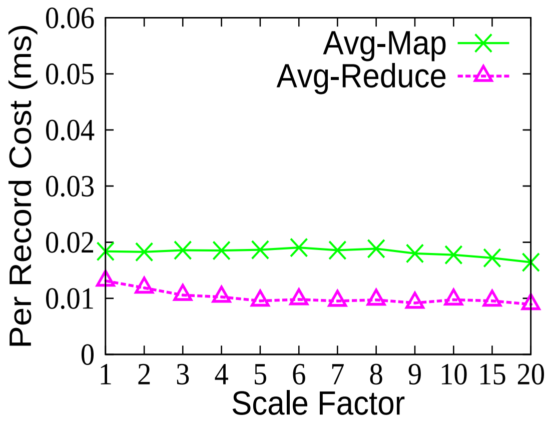
Using these job profiles we predict T^{avg} for *TPC-H Q10* based on different scale factors. The predicted results are accurate for scale factors 10,15, 20, and still acceptable for scale factor 7. The prediction errors for scale factors 3 and 5 are significantly higher.

To understand the logic behind these results, we analyze *TPC-H Q10* processing in more detail (see its DAG in Figure 3.17 (c)). Our measurements reveal that the first stage is responsible for 40% of the overall execution time. Therefore, analyzing the profiles of two concurrent jobs J_1 and J_2 of this stage will be very useful.

Figures 3.25 (a) and (b) show the processing costs per record during different execution phases of J_1 and J_2 respectively. Apparently, the job profiles are quite stable starting at the scale factor 8 and up. It explains why the completion time predictions for these scale factors are accurate. However, for small scale factors, the processing costs per record are significantly higher. There is an overhead associated with a task execution and it contributes a significant portion in the task duration when processing a small set of records. The overhead impact is significantly diminished when processing a larger dataset, and the cost per record becomes more representative of the actual processing cost.



(a) Job profiles for Q10-1-1



(b) Job profiles for Q10-1-2

Figure 3.25: Profile for *TPC-H Q10* with different input data size (scale factor).

3.4.2 Impact of input data on the map function performance

As we discussed in Chapter 3.1, the map (reduce) task is consists of a sequence of phases, and among these phases, there are two customized ones: the map and reduce phases. There execution performance is determined by the user defined functions and we estimate the duration for these customized phases by extracting a profile from the past execution which contains the average execution time per record for the map(reduce) functions and use the profile to estimate the map(reduce) phase duration when it is execution on a larger data set.

The intuition behind the approach is that the map(reduce) function performs the same computation logic on each input record and the average execution time per record for the map(reduce) function remains consistent when applied on different input data set. Such assumption holds for most data processing operation such as *projection* and *selection* and simple MapReduce application such as *WordCount* and *Sort* However, for some more complex applications, the map(reduce) function performance could be significantly impacted by the input data set.

As an example, the computation of the map function in the *KMeans* application significantly depends on the number of initial centroids. It defines the number of clusters in the clustering algorithm (i.e., the K value). Table 3.7 shows the processing time per record of the map function for *KMmeans* with different number of initial centroids (i.e., *K=16* and *K=50*).

Table 3.7: Processing time per record for *KMeans* with different number of initial centroids.

	Process time per record
KMeans_16	175ms
KMeans_50	522ms

For *KMeans* with 50 centroids, the map function has a much higher execution time compared with *KMeans* with 16 centroids, since the increased number of centroids in the clustering algorithm increases the number of comparisons for each record in the map phase and it leads to an increased compute time of the map function. As a result, for these complex application whose map(reduce) function impacted by input data, we have to extract the profile based on the new data set and use it for a more accurate performance prediction.

3.4.3 Impact of data skew in reduce stage

The proposed MapReduce performance model relies on the assumption that the intermediate data generated by the map stage is uniformly distributed across the reduce tasks. However, this assumption may not hold for some applications with skewed input/output data.

Take the *KMeans* application again for example, Figure 3.11 in Chapter 3.2.3 shows the predicted completion times closely approximate the measured ones. In these experiments, we considered the *KMeans* application with *K=50*, i.e., defined by the number of reduce tasks has set to 50. The performance prediction for *K-Means* application with *K=50* is quite accurate.

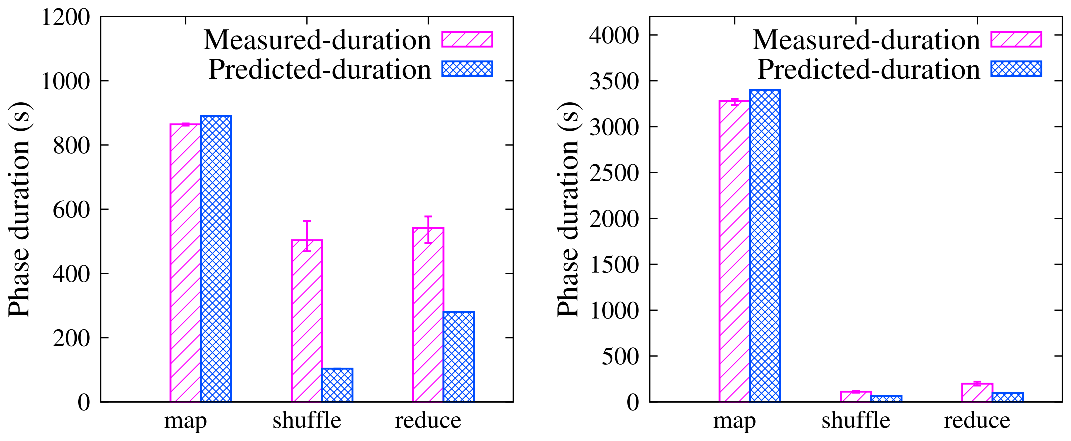
However, the situation changes when we perform the same experiments for *KMeans* with *K=16*. Table 3.8 shows the measured and predicted completion time for the *KMeans_16* and *KMeans_50* when these jobs are executed on the large 66-node cluster.

Table 3.8: Measured and predicted completion times for *KMmeans*

	Measured Completion Time (sec)	Predicted Completion Time (sec)
KMeans_16	1910	1275
KMeans_50	3605	3683

The measured completion time for *KMmeans_16* is 1910 sec. while the predicted completion time is 1275 sec (i.e., a prediction error of 33%). For a more detailed analysis of the *KMeans* execution time under different parameters, we break down its overall completion time into three main stages, such as *map*, *shuffle*, and *reduce* stages, and compare their durations when it is configured with 16 and 50 reduce tasks respectively. The results are illustrated in Figure 3.26.

We can see that the proposed model predicts accurately the duration of the map stage in both cases: the difference between the measured and predicted results is 3% and 4% respectively. However, for *KMeans* with 16 reduce tasks, the model under-estimates the shuffle and reduce stage durations. Since the shuffle and reduce stages represent a significant fraction in the overall completion time of *KMeans_16* – this leads to a significant inaccuracy in predicting the job completion time. The prediction error is caused by the skew in the intermediate data and the unbalanced data distribution to the reduce tasks. As the number of input centroids (i.e., the *K* value) also defines the number of reduce tasks, by increasing the reduce task number (the number of initial centroids), the amount of data attached



(a) *KMeans_16* (b) *KMeans_50*

Figure 3.26: Predicted vs measured stage durations for *KMeans* application with different number of reduce tasks (i.e., $K = 16$ and $K = 50$).

with each key is more evenly distributed which reduced the unbalanced work in reduce stage. Moreover, the increased number of reduce tasks results in a smaller portion of data processed by each reduce task. This significantly decreases the durations of shuffle and reduce stages and masks a possible impact of a data skew in these stages on the overall completion time. Therefore, the prediction of execution time for *KMmeans_50* is more accurate.

3.4.4 Variability of job profiles in public cloud environment

Another interesting problem is to understand the variability of job profiles. If the job profiles when executed on the MapReduce framework vary significantly across different executions, it will lead to inaccuracy with our performance modeling framework in predicting the completion times. The question becomes more important in public cloud environment as it is expected that there will be more performance uncertainty in such shared environment.

We execute the set of 13 applications shown in Table 3.2 on three Hadoop clusters deployed with different types of EC2 VM instances: *small, medium* and *large* with more details about each instance type and our configurations in Table 3.4 in Chapter 3.2.3

Tables 3.9-3.11 summarize the job profiles collected for these applications. For better analysis, we separate the shuffle duration from the reduce task as the shuffle duration is mostly affected by the network performance. Specifically, these tables show the average durations for map, shuffle and reduce processing as well as the standard deviation for these stages across 5 runs.

The standard deviation for the measured map, shuffle, and reduce processing shows that the map task durations are very stable across different applications and clusters (stdev is less than 10% for all cases). While shuffle durations for some applications exhibit a higher standard deviation. From one hand, the variability is caused by the shared network where different instances compete for the same bandwidth. One the other hand, the data skew also caused more variability as the shuffle time is significantly impacted by the amount of data that needs to be transferred. As a result, the *Classification* and *KMeans* applications show a higher stand deviation in their shuffle processing. The reduce processing turns to be more stable than the shuffle processing. However, some of the applications (e.g., *Grep*, *HistMovies* and *HistRatings*), since they contain a single reduce task, their reduce processing shows a higher stdev mostly because the different performance they have when executed on different instances.

One interesting observation from the analysis of the job profiles is that the shuffle durations of the Hadoop cluster formed with *large* instances are much longer compared to the clusters formed with *small* instances. The reason is that the Amazon EC2 instance scaling is done with respect to the CPU and RAM capacity, while the storage and network bandwidth is only fractionally improved. As we configure a higher number of slots on *large* instances, it increases the I/O and network contention among the tasks running on the same instance, and it leads to significantly increased durations of the shuffle phase. At the same time, the map task durations of most applications executed on the Hadoop cluster with *large* instances are significantly improved, e.g., the map task durations of *Classification* and *KMeans* applications improved almost three times. We will discuss about the problem and possible opportunities from these observations in Chapter 4.2.

Table 3.9: Job profiles on the EC2 cluster with small instances (time in sec)

Application	*avgMap*	*avgShuffle*	*avgReduce*	*map* STDEV	*shuffle* STDEV	*reduce* STDEV
TeraSort	29.1	248.5	31.2	0.82%	4.51%	0.97%
WordCount	71.5	218.7	12.1	1.16%	5.83%	3.68%
Grep	19.0	125.7	4.5	1.19%	26.43%	10.53%
InvIndex	83.9	196.8	18.2	1.33%	8.03%	3.96%
RankInvIndex	35.4	376.0	81.9	1.05%	3.79%	0.81%
TermVector	98.9	360.0	137.2	0.78%	2.45%	2.45%
SeqCount	101.2	256.8	54.1	1.01%	3.63%	6.62%
SelfJoin	11.9	217.9	12.3	0.70%	4.87%	3.12%
AdjList	265.9	72.7	291.1	1.53%	6.57%	0.84%
HistMovies	17.9	138.9	3.4	1.49%	40.85%	34.84%
HistRating	58.9	111.8	4.8	2.10%	35.58%	22.41%
Classif	3147.3	58.5	4.0	1.21%	12.76%	3.13%
Kmeans	3155.9	80.4	87.5	0.32%	30.09%	11.43%

Table 3.10: Job profiles on the EC2 cluster with medium instances (time in sec)

Application	*avgMap*	*avgShuffle*	*avgReduce*	*map* STDEV	*shuffle* STDEV	*reduce* STDEV
TeraSort	36.9	466.3	26.5	1.06%	14.07%	1.21%
WordCount	83.0	562.4	11.6	0.48%	7.01%	9.09%
Grep	23.8	256.6	3.2	4.95%	24.13%	9.48%
InvIndex	101.0	449.5	13.6	0.52%	8.65%	1.62%
RankInvIndex	45.7	741.6	64.0	0.63%	9.40%	2.77%
TermVector	128.1	432.4	71.9	0.23%	7.08%	2.81%
SeqCount	126.8	482.1	35.0	0.52%	21.70%	14.98%
SelfJoin	11.1	408.1	11.2	0.92%	13.86%	1.65%
AdjList	270.1	163.2	206.4	2.74%	8.70%	1.16%
HistMovies	20.1	246.7	3.7	3.14%	26.39%	17.04%
HistRating	71.7	240.4	5.0	0.23%	31.39%	14.22%
Classif	3013.8	177.2	3.9	0.82%	44.03%	4.33%
Kmeans	2994.0	189.7	51.7	3.93%	80.84%	6.96%

3.5 Conclusion

Hadoop is increasingly being deployed in enterprise private clouds and also offered as a service by public cloud providers (e.g., Amazons Elastic Map-Reduce). Many companies are embracing Hadoop for advanced data analytics over large datasets that require completion time guarantees. Design of new job profiling tools and performance models for MapReduce environments has been an active research topic in industry and academia during past few years.

Table 3.11: Job profiles on the EC2 cluster with large instances (time in sec)

Application	*avgMap*	*avgShuffle*	*avgReduce*	*map* STDEV	*shuffle* STDEV	*reduce* STDEV
TeraSort	27.3	806.4	20.0	0.66%	7.78%	16.14%
WordCount	54.7	1028.6	12.9	4.33%	10.24%	9.15%
Grep	18.3	791.8	4.3	3.50%	16.48%	22.81%
InvIndex	61.8	1152.6	14.9	6.47%	5.10%	8.68%
RankInvIndex	28.3	1155.8	40.5	1.49%	9.20%	8.19%
TermVector	85.3	1007.6	30.2	3.88%	5.98%	10.04%
SeqCount	62.0	1046.1	37.6	1.51%	6.70%	2.10%
SelfJoin	16.4	1015.7	18.5	1.93%	4.86%	19.11%
AdjList	149.0	436.9	149.1	0.56%	13.34%	2.78%
HistMovies	22.3	724.2	5.2	6.97%	22.46%	17.25%
HistRating	51.4	628.6	3.6	10.59%	21.01%	40.83%
Classif	1004.6	711.2	3.9	0.87%	37.15%	27.74%
Kmeans	1024.6	716.9	58.5	1.31%	10.75%	5.25%

In this chapter, we offer a new MapReduce performance modeling framework that can efficiently predict the completion time of a MapReduce application. It combines 3 different performance models which includes 1) a platform performance model. 2) a MapReduce job performance model and 3) a MapReduce workflow performance model. We first use a set of microbenchmarks to profile generic phases of the MapReduce processing pipeline of a given Hadoop cluster and derive an accurate platform performance model of a given cluster. Next, the introduced MapReduce job performance model combines the knowledge of the extracted job profile and the derived platform performance model to predict a MapReduce job completion time on a new dataset. Finally, the workflow performance model is used to predict the completion time of a MapReduce workflow that could contain both sequential and concurrent jobs. Our approach is **non-intrusive, efficient** and **accurate**. The proposed approach also enables automated deadline-driven resource allocation and provisioning for complex MapReduce applications defined by the DAGs of MapReduce jobs which will be covered in the next chapter.

Chapter 4

Resource Management for MapReduce Applications

It is common in the enterprise setting, that a Hadoop cluster is shared by multiple applications and each of these applications need to complete with certain time target, that are formulated as the completion time guarantees. The technological trend towards using MapReduce based frameworks in support of these latency-sensitive applications requires the system to employ an automatic resource allocation control for achieving different performance goals for each application. Currently, there is no job scheduler for MapReduce environments that given a job completion deadline, could allocate the appropriate amount of resources to the job so that it meets the deadline.

On the other hand, the advent of cloud computing provides a new delivery model with virtually unlimited computing and storage resources. It offers a compelling alternative to rent resources in a "pay-as-you-go" fashion. It is an attractive and cost-efficient option for many users because acquiring and maintaining a complex, large-scale infrastructure such as a Hadoop cluster requires a significant upfront investment and then a continuous maintenance and management support. However, a typical cloud environment offers a choice of different capacity Virtual Machines for deployment with different prices. These VMs can be deployed on a variety of hardware and be allocated different amounts of system resources.

Therefore, a user is facing a variety of platform and different choices could lead to significant difference in execution performance as well as the monetary cost depending on the applications the user plans to execute. The selection of the optimal cluster deployment is a non-trivial problem and currently there is no guidance that could help the user make the decisions.

In this chapter, we focus on the resource management for MapReduce workloads in order to achieve their performance goals. Specifically, we try to provide solutions to the following two problems.

- *Resource allocation in shared Hadoop cluster.* i.e., given a MapReduce application which could be defined as a DAG of MapReduce jobs with a completion time goal, determine the appropriate amount of resources required for completing it with a given (*soft*) deadline and control the resource allocation with our deadline-aware scheduler.

- *Resource provision in public cloud environment.* i.e., given a workload that consists of multiple MapReduce applications, select the type and size of the underlying platform for a Hadoop cluster that provides best cost/performance trade-offs: i) minimizing the cost (budget) while achieving a given makespan target, or ii) minimizing the achievable makespan for a given budget.

4.1 Deadline-driven resource allocation on shared Hadoop cluster

We first propose our solution to the deadline driven resource allocation problem for MapReduce applications. It is based on the performance evaluation model described in Chapter 3 and contains an efficient strategy to estimate the minimal resource requirement for an application to achieve its completion time target. We start, as a building block, the resource management for applications that contain a single MapReduce job and then extend it for more complex applications defined as MapReduce workflows. We first propose a simple approach which works effectively for sequential MapReduce workflows, and then refine it to incorporate the execution overlaps for concurrent jobs within a workflow.

Besides, in refining the resource allocation approach, we identify that the execution order of the concurrent jobs within a workflow could significantly affect the overall completion time. Motivated by such observation, we first optimize a MapReduce workflow execution by enforcing the *optimal* schedule of its concurrent jobs. The proposed optimization could reduce the total completion time. Moreover, it has another useful outcome: it eliminates existing non-determinism in execution of concurrent jobs, and therefore, it enables better performance predictions and more accurate resource estimates.

4.1.1 Resource allocation for single MapReduce job

As a building block, we first introduce the resource allocation strategy for simple latency sensitive applications that contain a single MapReduce job. It is based on the bounds based performance model as described in Chapter 3.2.2. As the proposed model predicts the job completion time as a function of allocated resources (i.e., the number of map and reduce slots) using the following form:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low} \tag{4.1}$$

It also can be used to find the appropriate number of map and reduce slots that could support a given job deadline D: let us substitute D instead of T_J^{low} in Equation 4.1 as

$$D = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low} \tag{4.2}$$

Equation 4.2 yields a hyperbola if S_M^J and S_R^J are the variables. Figure 4.1 shows an example of such hyperbola.

All integral points on this hyperbola are possible allocations of map and reduce slots which result in meeting the same deadline D. There is a point where the sum of the required map and reduce slots is minimized. We calculate this minima on the curve using Lagrange's multipliers [66], since we would like to conserve the number of map and reduce slots required for the minimum resource allocation per

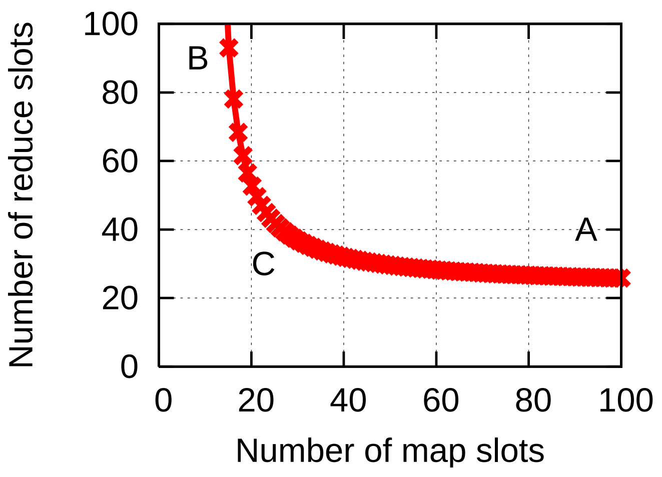


Figure 4.1: Resource allocations satisfy a given deadline

job J with a given deadline D. Note, that we can use D for finding the resource allocations from the corresponding equations for upper and lower bounds on the job completion time estimates.

4.1.2 Resource allocation for MapReduce workflows: a basic approach

Now, consider a more complex application defined as a workflow that consists of N jobs: $W = \{J_1, J_2, ...J_N\}$ with a given completion time goal D. The problem is is to estimate a required resource allocation (a number of map and reduce slots) that enables the workflow W to be completed with the (soft) deadline D.

First of all, there are multiple possible resource allocations that could lead to a desirable performance goal. We could have picked a set of intermediate completion times D_i for each job J_i from the set $W = \{J_1, J_2, ...J_N\}$ such that $D_1 + D_2 + ... + D_N \leq D$, and then determine the number of map and reduce slots required for each job J_i to finish its processing within D_i. However, such a solution would be difficult to implement and manage by the scheduler. When each job in a DAG requires a different allocation of map and reduce tasks then it is difficult to reserve and guarantee the timely availability of the required resources.

A simpler and more elegant solution would be to determine a specially tailored resource allocation of map and reduce slots (S_M^W, S_R^W) to be allocated to the entire workflow W (i.e., to each its job J_i, $1 \leq i \leq N$) such that W would finish within a given deadline D. We called it the *basic* resource allocation approach.

There are a few design choices for determining the required resource allocation for a given MapReduce workflow. These choices are driven by the bound-based performance models designed in Chapter 3.2.2:

- Determine the resource allocation when deadline D is targeted as a *lower bound* of the workflow completion time. Typically, this leads to the least amount of resources that are allocated to the workflow for finishing within deadline D. The lower bound on the completion time corresponds to "ideal" computation under allocated resources and is rarely achievable in real environments.
- Determine the resource allocation when deadline D is targeted as an *upper bound* of the workflow completion time. This would lead to a more aggressive resource allocations and might result in a workflow completion time that is much smaller (better) than D because worst case scenarios are also rare in production settings.
- Finally, we can determine the resource allocation when deadline D is targeted as the *average* between lower and upper bounds on the workflow completion time. This solution provides a balanced resource allocation that is closer for achieving the workflow completion time D.

For example, when D is targeted as a *lower bound* of the workflow completion time, we need to solve the following equation for an appropriate pair(S_M^W, S_R^W) of map and reduce slots:

$$\sum_{1 \leq i \leq N} T_{J_i}^{low}(S_M^W, S_R^W) = D \tag{4.3}$$

By using the Lagrange's multipliers method as described in [66], we determine the minimum amount of resources (i.e. a pair of map and reduce slots (S_M^W, S_R^W) that results in the minimum sum of the map and reduce slots) that needs to be allocated to W for completing with a given deadline D.

Solution when D is targeted as an *upper bound* or an *average* between lower and upper bounds of the workflow completion time can be found in a similar way.

Evaluating the basic approach in supporting deadline-driven applications

We evaluate the accuracy of the basic approach in estimating the appropriate resource allocation for a MapReduce workflow with completion time requirement using the same PigMix benchmark and TPC-H and proxy query set described in Chapter 3.3.2 as well as the testing workloads. The experiments are performed on the same testbed as we described in Chapter 3.1.4.

We first evaluate the approach with the PigMix benchmark. In this set of experiments, let T denote the completion time when it is processed with maximum available cluster resources (i.e., when the entire cluster is used for processing). We set $D = 3 \cdot T$ as a completion time goal. Using the Lagrange multipliers' approach (described in Chapter 4.1.2) we compute the required resource allocation, i.e., a fraction of cluster resources, a tailored number of map and reduce slots that allow the workflow to be completed with deadline D. As discussed in Chapter 4.1.2, we can compute a resource allocation when D is targeted as either a lower bound, or upper bound or the average of lower and upper bounds on the completion time. Figure 4.2 shows the *measured* workflow completion times based on these three different resource allocations. Similar to our earlier results, for presentation purposes, we **normalize** the achieved completion times with respect to deadline D.

In most cases, the resource allocation that targets D as a lower bound is insufficient for meeting the targeted deadline (e.g., the L17 program misses deadline by more than 20%). However, when we compute the resource allocation based on D as an upper bound – we are always able to meet the required deadline, but in most cases, we over-provision resources, e.g., L16 and L17 finish more than 20% earlier than a given deadline. The resource allocations based on the average between lower and upper bounds result in the closest completion time to the targeted deadlines.

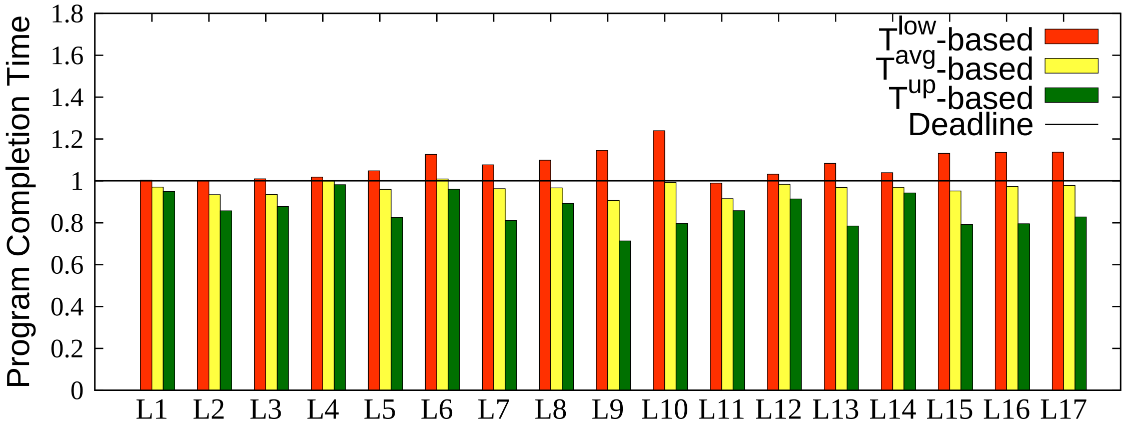
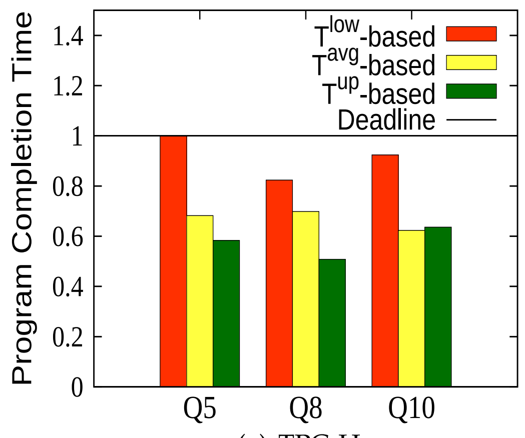


Figure 4.2: PigMix executed with the estimated resources: do we meet deadlines?

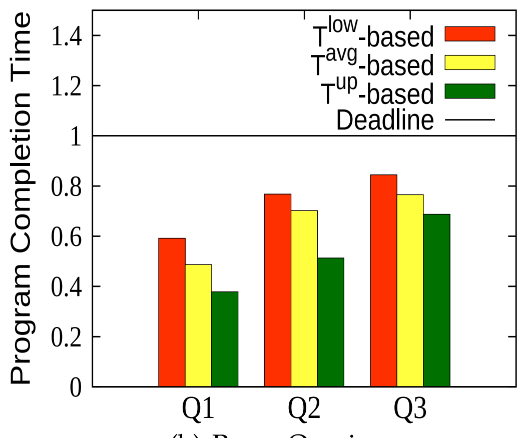
The basic approach proves to be effective for the MapReduce workflows generated from the PigMix benchmark. However, as most of the queries from PigMix are compiled into sequential MapReduce workflow, it is not clear whether the proposed approach works well for workflows with concurrent jobs. To understand the performance of the approach for workflows with concurrent jobs. We performance the similar experiments on the two other workloads: TPC-H and Proxy query set.

Figure 4.3 presents the results for these two workloads. While each of the three considered resource allocations is meeting the desired deadline, we observe that the basic approach is inaccurate for programs with concurrent jobs. There is significant resource over-provisioning: the considered workflows finish much earlier (up to 50% earlier) than the targeted deadlines.

In summary, while the basic approach produces good results for workflows with sequential MapReduce jobs, it over-estimates a completion time of workflows with concurrent jobs, and leads to over-provisioned resource allocations for workflows with concurrent jobs. The reason of the inaccuracy also comes from the execution overlaps among the concurrent jobs. As we discussed in Chapter 3.3.3. The pipelined execution of concurrent jobs in workflow W may significantly reduce



(a) TPC-H



(b) Proxy Queries

Figure 4.3: TPC-H/Proxy queries with the estimated resources: do we meet deadlines?

the program completion time. Therefore, W may need to be assigned a smaller amount of resources for meeting the same deadline D.

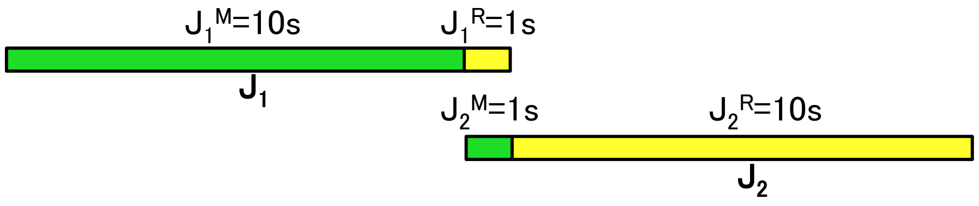
In the following part, we will first present an important observation we found that the execution order of the concurrent jobs could significantly affect the workflow completion time and propose a scheduling algorithm for optimizing the completion time based on the observation. We then refine the proposed approach for estimating the resource allocation of such optimized MapReduce workflows in meeting their deadlines.

4.1.3 Schedule concurrent jobs within a workflow

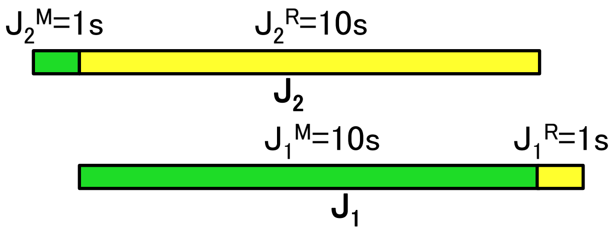
As we explained in Chapter 3.3.3, the concurrent jobs within a workflow are executed in a pipelined fashion which lead to shorter total completion time. With such execution model, we find one more interesting observation: the execution order of the concurrent jobs within a workflow could significantly affect the total completion time. The current Pig implementation submits concurrent MapReduce jobs from the same Pig program in a random order. Some ordering may lead to a significantly less efficient resource usage and an increased processing time. Consider the following example with two concurrent MapReduce jobs:

- Job J_1 has a map stage duration of $10s$ and the reduce stage duration of $1s$.
- Job J_2 has a map stage duration of $1s$ and the reduce stage duration of $10s$.

There are two possible executions of J_1 and J_2 shown in Figure 4.4: (a) J_1 is followed by J_2, (b) J_2 is followed by J_1.



(a) J_1 is followed by J_2.



(b) J_2 is followed by J_1.

Figure 4.4: Impact of concurrent job scheduling on their completion time.

Now, let us analyze the results of these two different execution orders in terms of the total completion time.

- *J_1 is followed by J_2*. Then, the reduce stage of J_1 overlaps with the map stage of J_2 leading to overlap of only $1s$. Thus, the total completion time of processing two jobs is $10s + 1s + 10s = 21s$.

- *J_2 is followed by J_1*. Then the reduce stage of J_2 overlaps with the map stage of J_1 leading to a much better pipelined execution and a larger overlap of 10s. Thus, the total makespan is $1s + 10s + 1s = 12s$.

As we can see, there can be a significant difference in the job completion time (75% in the example above) depending on the execution order of the jobs. To optimize the schedule of the concurrent jobs within a workflow, we apply the classic Johnson's algorithm for building the optimal two-stage jobs' schedule [35].

Johnson's Algorithm

In 1953, Johnson [35] proposed an optimal algorithm for a two stage production schedule. A collection of production items and two machines are given. Each item must pass through stage one, and then stage two. Each machine can handle only one item at a time. There are two arbitrary positive numbers given for each item representing the work time for that item to pass through the stage.

We restate the algorithm in terms of MapReduce jobs. Let us consider a collection of n jobs. Each job j is represented by the pair of map and reduce stage durations (m_j, r_j). Each job $j = (m_j, r_j)$ is augmented by the attribute D_j defined as follows:
If $min(m_j, r_j) = m_j$ then $D_j = (m_j, map)$ else $D_j = (r_j, reduce)$. The first argument in D_i called a *stage duration*, and the second the *stage type* (map or reduce).

An optimal schedule can be constructed by the following algorithm described below: it works by filling job indexes into the schedule σ by taking the the jobs from the list L and placing them into the schedule from the both ends (head and tail) and proceeding towards the middle. Some informal explanation of the algo-

Algorithm 1 Johnson's Algorithm

Input: List L of n MapReduce jobs. D_i is the stage duration and the stage type (map or reduce) of job i as defined above.
Output: Schedule σ : order of jobs

```
1:  Sort $L$ based on map or reduce stage durations using $D_i$
2:  $head \leftarrow 1, tail \leftarrow n$
3:  for each stage duration $D_i$ in $L$ do
4:      if $D_i$ is map stage then
5:          // Put job $i$ from the front
6:          $\sigma_{head} \leftarrow i$
7:          head ← head + 1
8:      else
9:          // Put job $i$ from the end
10:         $\sigma_{tail} \leftarrow i$
11:         tail ← tail - 1
12:     end if
13: end for
```

rithm. First, we order all the n jobs in the list L according to the following rule: job j precedes job $j+1$ if and only if $min(m_j, r_j) \leq min(m_{j+1}, r_{j+1})$. In other words, we can sort the jobs using their job attribute D_i that represents the stage duration (smallest duration of the two stages) and the stage type.

Then the jobs from this ordered list L put from the front of the schedule (if the duration represents the map stage) or from the end of the schedule (if the duration represents the reduce stage).

Let us illustrate the Johnson's algorithm execution with the following example workloads that contains five MapReduce jobs with description in Figure 4.5 (left part) where the last column represents the additional attribute D_i: This collection of jobs can be sorted according to the attribute D_i and Figure 4.5 (right part) shown the sorted set of MapReduce jobs.

i	m_i	r_i	D_i
1	4	5	(4,m)
2	4	1	(1,r)
3	30	4	(4,r)
4	6	30	(6,m)
5	2	3	(2,m)

i	m_i	r_i	D_i
2	4	1	(1,r)
5	2	3	(2,m)
1	4	5	(4,m)
3	30	4	(4,r)
4	6	30	(6,m)

Figure 4.5: Example of Johnson's Algorithm.

Then if we follow the Johnson's algorithm and start placing the jobs in the schedule from both ends toward the middle, we have the following sequence: (5,1,4,3,2). This job execution ordering define the schedule with minimum overall makespan. For our example, the makespan of optimal schedule is 47. The worst schedule is the reverse order of the optimal one: it has a makespan of 78 (this is 66% increase in the makespan compared to the optimal time). Indeed, the optimal schedule provides significant savings.

Evaluate the performance benefits with optimized concurrent job execution

We next evaluate the completion time improvements when we execute a workflow by enforcing the optimized execution order of its concurrent jobs. We use the TPC-

H and Proxy query set as the experiment workloads as they contain workflows with concurrent jobs.

Figures 4.6 and 4.7 show the scheduling impact of concurrent jobs on the workflow completion time for the three TPC-H queries $Q5, Q8$, and $Q10$ and Proxy queries $Q1, Q2$, and $Q3$ respectively when each workflow in those sets is processed with **128** map and **64** reduce slots.

Figures 4.6 (a) and 4.7 (a) show two extreme measurements: the best program completion time (i.e., when the optimal schedule of concurrent jobs is chosen) and the worst one (i.e., when concurrent jobs are executed in the "worst" possible order based on our estimates). For presentation purposes, the best (optimal) completion time time is *normalized* with respect to the worst one. The choice of optimal schedule of concurrent jobs reduces the completion time by 10%-27% compared with the worse case ordering.

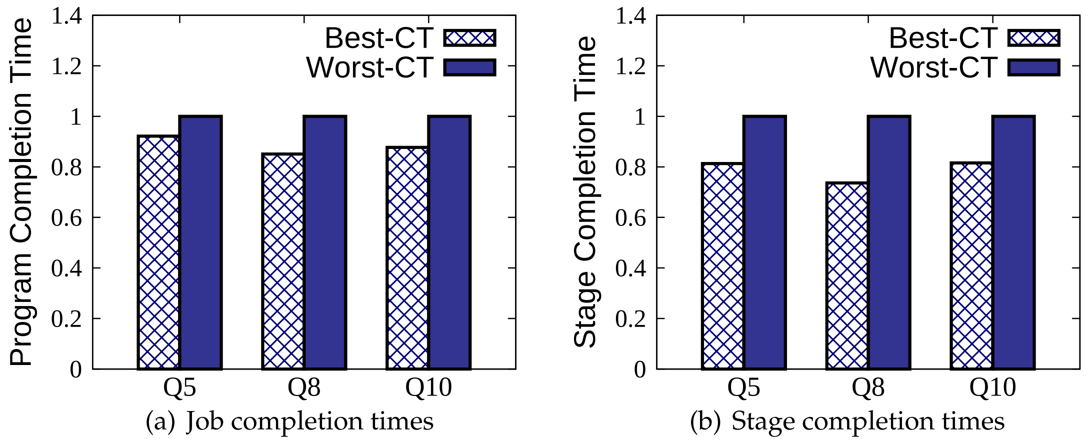
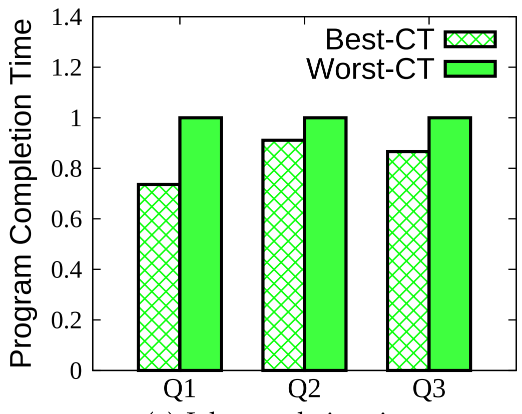


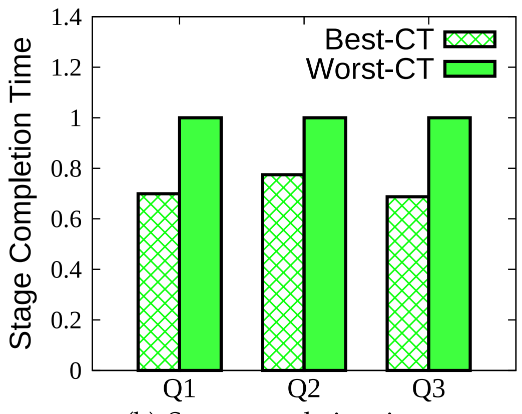
Figure 4.6: Measured completion times for different schedules of concurrent jobs in TPC-H queries.

The performance benefits with the optimized schedule of concurrent is even more pronounced if we consider the stage completion time. Figures 4.6 (b) and 4.7 (b) show completion times of stages with concurrent jobs under different schedules for the same TPC-H and Proxy queries. The performance benefits at the stage level are even higher: they range between 20%-30%.

In summary, the optimal execution of concurrent jobs leads to a better overall



(a) Job completion times



(b) Stage completion times

Figure 4.7: Measured completion times for different schedules of concurrent jobs in Proxy queries.

performance: an improved completion time and better resource utilization. Moreover, this optimization has another useful outcome: it eliminates possible non-determinism in workflow execution by eliminating the random execution order of concurrent jobs. This enables a more accurate performance model for a completion time prediction and a *refined* resource allocation approach for estimating the appropriate resource requirements of a given workflow in meeting its deadline.

4.1.4 Resource allocation for MapReduce workflows: a refined approach

Now, consider a MapReduce workflow W with a given deadline D, as we shown in the previous discussion, the optimized execution of concurrent jobs in W results in a shorter workflow completion time. Therefore, an even smaller amount of resources for W may satisfy the same given deadline D compared to its non-optimized execution. As shown in Chapter 4.1.2, the earlier proposed *basic* model does not take the pipelined execution between concurrent jobs in a workflow, and therefore, it overestimates the completion time and the amount of resources required for the workflows with concurrent jobs for meeting their deadlines. However, the unique benefit of this model is that it allows to express the completion time D of a workflow via a special form equation shown below:

$$D = \frac{A^W}{S_M^W} + \frac{A^W}{S_R^W} + C^W \tag{4.4}$$

where S_M^W and S_R^W denote the number of map and reduce slots assigned to W. As we show Chapter 4.1.2, Equation eq. (4.4) can be used for finding the resource allocation (S_M^W, S_R^W) such that the workflow W completes within time D. This equation yields a hyperbola if S_M^W and S_R^W are considered as variables. We can directly calculate the minima on this curve using Lagrange's multipliers (see Chapter 3.2.2).

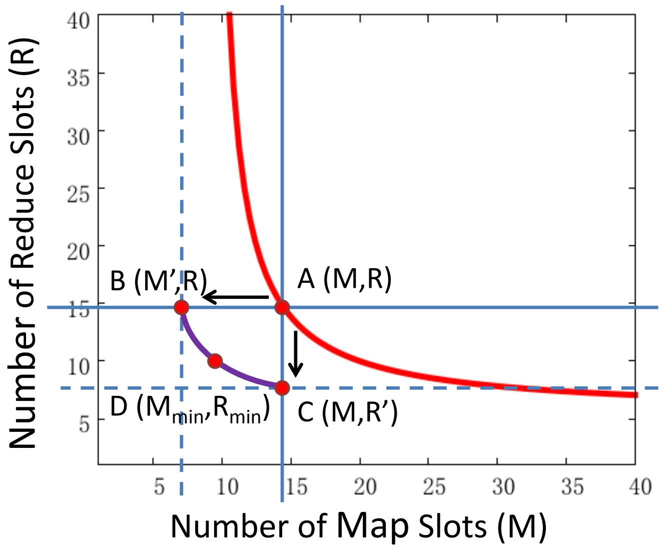


Figure 4.8: Resource allocation estimates for an optimized Pig program.

The performance model introduced in Chapter 3.3.3 for accurate completion time estimates of a workflow with concurrent jobs is more complex. It requires computing a function *max* for stages with concurrent jobs, and therefore, it cannot be expressed as a single equation for solving the resource allocation problem. However, we can use the "over-sized" resource allocation derived by the *b*asic approach with eq. (4.4) as an initial point for determining the solution required by the optimized workflow W. The hyperbola with all the possible solutions according to the *basic* approach is shown in Figure 4.8 as the red curve, and $A(M, R)$ represents the point with a minimal number of map and reduce slots (i.e., the pair (M, R) results in the minimal sum of map and reduce slots). Algorithm 2 described below shows the computation for determining the minimal resource allocation pair (M_{min}, R_{min}) for an optimized Pig program P with deadline D. This computation is illustrated by Figure 4.8.

Algorithm 2 Determining the resource allocation for a Pig program

```
Input:
Job profiles of all the jobs in $W = \{J_1, J_2, ...J_N\}$
$D \leftarrow$ a given deadline
$(M, R) \leftarrow$ the minimum pair of map and reduce slots for $W$ and deadline $D$ by applying the basic model
Optimal execution of jobs $J_1, J_2, ...J_N$ based on $(M, R)$
Output:
Resource allocation pair $(M_{min}, R_{min})$ for optimized $W$

1: $M' \leftarrow M$, $R' \leftarrow R$
2: while $T_W^{avg}(M', R) \leq D$ do {   // From A to B}
3:     $M' \Leftarrow M' - 1$
4: end while
5: while $T_W^{avg}(M, R') \leq D$ do {   // From A to C}
6:     $R' \Leftarrow R' - 1$,
7: end while
8: $M_{min} \leftarrow M, R_{min} \leftarrow R$ , $Min \leftarrow (M + R)$
9: for $\hat{M} \leftarrow M' + 1$ to $M$ do {    // Explore blue curve B to C}
10:     $\hat{R} = R - 1$
11:     while $T_W^{avg}(\hat{M}, \hat{R}) \leq D$ do
12:         $\hat{R} \Leftarrow \hat{R} - 1$
13:     end while
14:     if $\hat{M} + \hat{R} < Min$ then
15:         $M_{min} \Leftarrow \hat{M}, R_{min} \Leftarrow \hat{R}, Min \leftarrow (\hat{M} + \hat{R})$
16:     end if
17: end for
```

First, we find the minimal number of map slots M' (i.e., the pair (M', R)) such that deadline D can still be met by the optimized workflow with the enforced optimal execution of its concurrent jobs. We do it by fixing the number of reduce slots to R, and then step-by-step reducing the allocation of map slots. Specifically, Algorithm 2 sets the resource allocation to $(M - 1, R)$ and checks whether workflow W can still be completed within time D (we use T_W^{avg} for completion time estimates). If the answer is positive, then it tries $(M - 2, R)$ as the next allocation. This process continues until point $B(M', R)$ (see Figure 4.8) is found such that the number M'

of map slots cannot be further reduced for meeting a given deadline D (lines 1-4 of Algorithm 2).

At the second step, we apply the same process for finding the minimal number of reduce slots R' (i.e., the pair (M, R')) such that the deadline D can still be met by the optimized Pig program P (lines 5-7 of Algorithm 2).

At the third step, we determine the intermediate values on the curve between (M', R) and (M, R') such that deadline D is met by the optimized MapReduce workflow W. Starting from point (M', R), we are trying to find the allocation of map slots from M' to M, such that the minimal number of reduce slots $\hat{R}$ should be assigned to W for meeting its deadline (lines 10-12 of Algorithm 2).

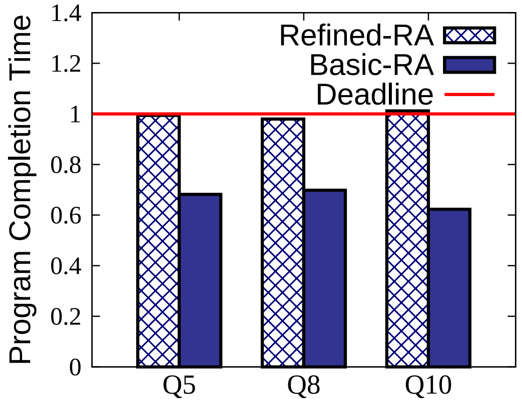
Finally, (M_{min}, R_{min}) is the pair on this curve such that it results in the the minimal sum of map and reduce slots.

Evaluate the refined approach in supporting deadline-driven applications

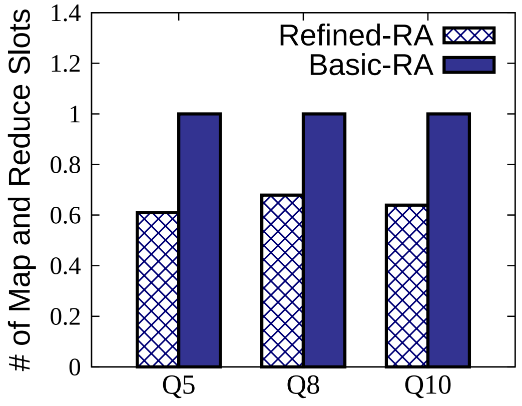
We evaluate the refined approach in estimating the appropriate resource allocation using the same TPC-H and Proxy query set as our experiment workloads as our *refined* approach focuses on latency sensitive MapReduce workflows with concurrent jobs. In this set of experiments, let T denote the total completion time when program W is processed with maximum available cluster resources. We set $D = 2 \cdot T$ as a completion time goal. Then we compute the required resource allocation for W by applying the refined resource allocation strategy when D is targeted as T_W^{avg}, i.e., the average of lower and upper bounds on the completion time.

Figures 4.9 (a) and 4.10 (a) compare the *measured* completion times achieved by the TPC-H and Proxy's queries respectively when they are assigned the resource allocations computed with the *basic* versus *refined* models. The completion times are **normalized** with respect to the targeted deadlines. While both models suggest sufficient resource allocations that enable the considered workflows to meet their deadlines, the resource allocations computed with the *refined* model are much more accurate: all the queries complete within 10% of the targeted deadlines.

Figures 4.9 (b) and 4.10 (b) compare the amount of resources (the sum of map

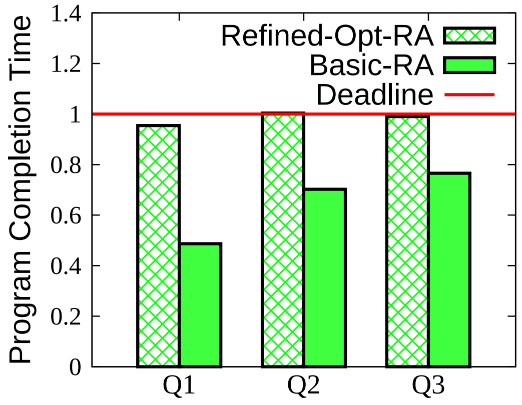


(a) Can we meet deadlines?

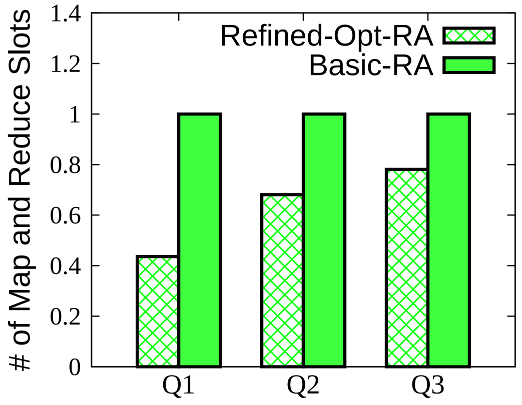


(b) Resource savings with *refined* approach

Figure 4.9: TPC-H Queries: efficiency of resource allocations with *refined* approach.



(a) Can we meet deadlines?



(b) Resource savings with *refined* approach

Figure 4.10: Proxy's Queries: efficiency of resource allocations with *refined* approach.

and reduce slots) computed with the *basic* approach versus *refined* approach for TPC-H and Proxy's queries respectively. The *refined* approach is able to achieve targeted deadlines with much smaller resource allocations (20%-40% smaller) compared to resource allocations suggested by the *basic* approach. Therefore, the proposed optimal schedule of concurrent jobs combined with the *refined* resource allocation strategy lead to the efficient execution and significant resource savings for deadline-driven MapReduce applications with concurrent jobs.

4.1.5 Deadline-driven job scheduler

Based on the estimated resource allocation for each application, our ultimate goal is to propose a novel deadline-drive scheduler for MapReduce environments that supports a new API: a MapReduce application can be submitted with a desirable completion time target (deadline). The scheduler will then estimate and allocate the appropriate number of map and reduce slots to the job so that it meets the required deadline. Figure 4.11 shows the implementation of our scheduler. Specifically, it consists of the following five interacting components shown in Figure 4.11:

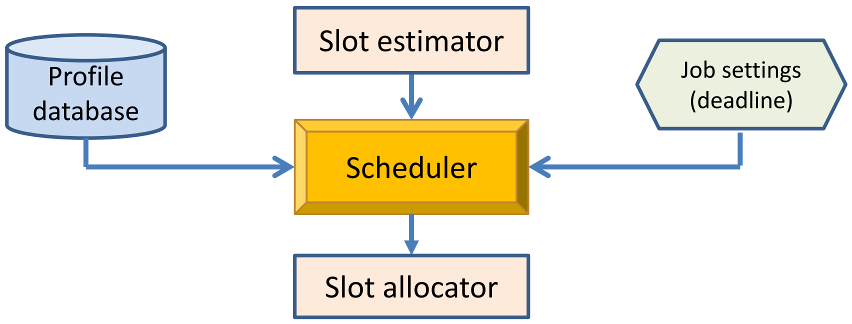


Figure 4.11: Implementation of the deadline-scheduler.

1. **Profile Database:** We use a MySQL database to store the past profiles extracted for each job. The profiles are identified by the (user, job name) which can be specified by the application.
2. **Slot Estimator:** Given the past profile of the job and the deadline, the slot estimator calculates the minimum number of map and reduce slots that need to be allocated to the job in order to meet its deadline. Essentially, it uses the refined approach introduced in Chapter 4.1.4.
3. **Slot Allocator:** Using the slots calculated from the slot estimator, the slot allocator assigns tasks to jobs such that the job is always below the allocated thresholds by keeping track of the number of running map and reduce tasks. In case there are spare slots, they can be allocated based on the additional policy. There could be different classes of jobs: jobs with/without deadlines. We envision that jobs with deadlines will have higher priorities for cluster resources than jobs without deadlines. However, once jobs with deadlines are allocated their required minimums for meeting the SLOs, the remaining slots can be distributed to the other job classes.

4. **SLO-Scheduler:** This is the central component that co-ordinates events between all the other components. Hadoop provides support for a pluggable scheduler. The scheduler makes global decisions of ordering the jobs and allocating the slots across the jobs. The scheduler listens for events like job submissions, worker heartbeats, etc. When a heartbeat containing the number of free slots is received from the workers, the scheduler returns a list of tasks to be assigned to it.

The scheduler has to answer two inter-related questions: which job should the slots be allocated and how many slots should be allocated to the job? The scheduler executes the *Earliest Deadline First* algorithm (EDF) for ordering the applications to maximize the utility function of all the users. For applications that defined as MapReduce workflows, the scheduler also enforces the optimized execution order for the concurrent branches within its workflow. The second question is answered using the refined approach discussed in Chapter 4.1.4. The detailed slot allocation schema is shown in Algorithm 3.

As shown in Algorithm 3, it consists of two parts: 1) when an application is added, and 2) when a heartbeat is received from a worker. Whenever an application is added, we fetch the profiles of all jobs that belongs to the application from the database and compute the minimum number of map and reduce slots required to complete the job within its specified deadline using our refined resource allocation estimates discussed earlier in Chapter 4.1.2.

Workers periodically send a heartbeat to the master reporting their health, the progress of their running tasks and the number of free map and reduce slots. In response, the master returns a list of tasks to be assigned to the worker. The master tracks the number of running and finished map and reduce tasks for each job. For each free slot and each job, if the number of running maps is lesser than the number of map slots we want to assign it, a new task is launched. As shown in Lines 9 - 13, preference is given to tasks that have data local to the worker node. Finally, if at least one map has finished, reduce tasks are launched as required.

Algorithm 3 Earliest deadline first algorithm

```
1: When application $W$ is added:
2: Fetch the profiles for each job within $W$ from database
3: Compute the execution order for concurrent jobs (if any) within $W$
4: Compute minimum number of map and reduce slots $(m_w, r_w)$ using our re-
   fined approach
5: When a heartbeat is received from node $n$:
6: Sort $workflows$ in order of earliest deadline
7: for each slot $s$ in free map/reduce slots on node $n$ do
8:    for each $w$ in $workflows$ do
9:       if $RunningMaps_w < m_w$ and $s$ is map slot then
10:         get the next ready job $j$ within $w$
11:         if job $j$ has unlaunched map task $t$ with data on node $n$ then
12:            Launch map task $t$ with local data on node $n$
13:         else if $j$ has unlaunched map task $t$ then
14:            Launch map task $t$ on node $n$
15:         end if
16:      end if
17:      if $RunningReduces_w < r_w$ and $s$ is reduce slot then
18:         get the next ready job $j$ within $w$
19:         if job $j$ has unlaunched reduce task $t$ then
20:            Launch reduce task $t$ on node $n$
21:         end if
22:      end if
23:   end for
24: end for
25: for each task $T_j$ within $W$ finished do
26:    Recompute $(m_w, r_w)$ based on the current time, current progress and dead-
       line of $W$
27: end for
```

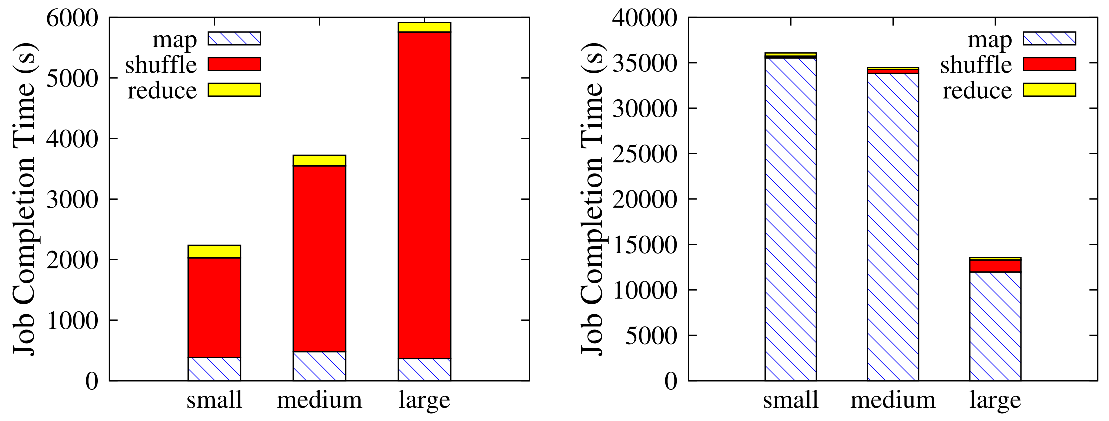
In some cases, the amount of slots available for allocation is less than required minima for job J and then J is allocated only a fraction of required resources. As time progresses, the resource allocations are recomputed during the job's execution and adjusted if necessary as shown in Lines 22-24.

4.2 Resource provisioning in public cloud environment

Cloud computing provides a new delivery model with virtually unlimited computing and storage resources. It offers a compelling alternative to rent resources in a ”pay-as-you-go” fashion. It is an attractive and cost-efficient option for many users because acquiring and maintaining a complex, large-scale infrastructure such as a Hadoop cluster requires a significant up-front investment and then a continuous maintenance and management support.

A typical cloud environment offers a choice of different capacity Virtual Machines for deployment with different prices. For example, the Amazon EC2 platform provides a choice of *small, medium,* and *large* VM instances, where the CPU and RAM capacity of a *medium* VM instance is two times larger than a capacity of a *small* VM instance, and the CPU and RAM capacity of a *large* VM instance is two times larger than a capacity of a *medium* VM instance. These differences are also reflected in the pricing: the *large* instance is twice (four times) more expensive compared with the *medium (small)* instance. It means that for the same price a user may deploy a 40-node Hadoop cluster using 40 *small* VM instances (with one map and one reduce slot per instance) or a 10-node Hadoop cluster using 10 *large* VM instances (with four map and four reduce slots per instance). Therefore, a user is facing a variety of platform and configuration choices that can be obtained for the same budget. Intuitively, these choices may look similar, and it might not be clear whether there is much difference in MapReduce application performance when these applications are executed on different type platforms.

Figure 4.12 shows two motivating examples. In these experiments, we execute two popular applications *TeraSort* and *KMeans* on three Hadoop clusters deployed with different type VM instances: *i)* 40 *small* VMs, *ii)* 20 *medium* VMs, and *iii)* 10 *large* VM instances. We configure Hadoop clusters according to their nodes capacity: each *small* VM instance is configured with one map and one reduce slot per instance; similarly, *medium* (*large*) VM instances are configured with two (four) map and two (four) reduce slots per instance. Therefore, these clusters can be obtained for the same price per time unit.



(a) TeraSort (b) KMeans

Figure 4.12: Completion time of two applications when executed on different type EC2 instances.

Apparently, a Hadoop cluster with 40 *small* instances provides the best completion time (so as the monetary cost) for a *TeraSort* application as shown in Fig. 4.12 (a), while a Hadoop cluster with 10 *large* instances is the best option for *KMeans* as shown in Fig. 4.12 (b). To further understand different stage contributions into the overall job completion time, we present the completion time break-down with respect to the map/shuffle/reduce stage durations. The results show that *TeraSort* has a longer shuffle duration when executed on large(medium) instances then the medium(small) ones which lead to longer completion time. The reason is as we explained in Chapter 3.4.4 that cloud environments use VM instance scaling with respect to CPU and RAM capacity but not with the storage and network bandwidth As we configure more slots on *large* instances, it increases the I/O and network contention that leads to significantly increased durations of the shuffle phase (The job profiles shown in Tables 3.9-3.11 also confirms our explanation). On contrast, for *KMeans*, the map stage duration dominates and the map execution is significantly improved when executed on large instances. On explanation is as shown in Table 4.1, most large instances are configured with a different (more powerful) CPU model than the small and medium ones. [1]

[1] We reserved 30 instances for each type and gather their cpu information.

Instance type	*CPU type*
Small	30/30 Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz
Medium	19/30 Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz 11/30 Intel(R) Xeon(R) CPU E5430 @ 2.66GHz
Large	26/30 Intel(R) Xeon(R) CPU E5-2651 v2 @ 1.80GHz 4/30 Intel(R) Xeon(R) CPU E5-2650 @ 2.00GHz

Table 4.1: CPU types used by different EC2 instances.

As a result, for applications like *TeraSort* which has a longer shuffle stage, the *small* instances outperforms the *large* ones. While for applications such as *KMeans* whose map stage duration dominates in the total completion time, it get more benefits when executed on the Hadoop cluster with *large* instances. These examples demonstrate that seemingly equivalent platform choices for a Hadoop cluster might result in a different application performance.

In this chapter, we aim to solve the **problem of the platform choice** to provide the best cost/performance trade-offs for a given MapReduce workload. As shown in Fig. 4.12 this choice is non-trivial and depends on the application characteristics. The problem becomes even more challenging for MapReduce workload with performance objectives to minimize the makespan (the overall completion time) of the given job set.

For a given a set of jobs J we aim to offer a framework for solving the following two problems:

- given a customer makespan target T (i.e., a desirable completion time for the entire set of jobs J), select the instance type, the cluster size, and propose the job schedule for meeting the makespan target T while minimizing the cost;
- given a customer budget C, select the instance type, the cluster size, and propose the job schedule, that minimizes achievable makespan for a given cost C.

4.2.1 Solution framework

The performance modeling framework could accurately estimate the completion time of a given MapReduce application according to the allocated resource. The framework also forms the foundation of our resource allocation strategy that implemented with our deadline-driven scheduler. However, the framework is not accurate enough for our resource provision problem. From our discussion in Chapter 3.3.3, we already observed that there are execution overlaps between the map and reduce stage across different jobs. However, in practical execution, the map (reduce) stages from different jobs could also be interleaved if there are enough available resources or if the jobs contains small number of map(reduce) tasks (e.g., the Grep application only has a single reduce task). The modeling framework does not consider such overlap which leads to pessimistic results in those cases. Moreover, for a workload that contains multiple independent MapReduce applications, the inaccuracy will accumulate when more jobs are executed at the same time.

To address the problem, we make use of a new MapReduce simulator, called *SimMR* [67], that can replay execution traces of real workloads collected in Hadoop clusters (as well as synthetic traces based on statistical properties of workloads) for evaluating different resource allocation and scheduling ideas in MapReduce environments.

Specifically, the designed framework is based on the following three main components:

- *Job Profiler:* it extracts a detailed job processing trace that consists of N_M^J of map task durations and N_R^J shuffle/sort and reduce phase durations where N_M^J and N_R^J represent the number of map and reduce tasks within a job J. The job profile and processing trace can be obtained from the past run of this job [66] or extracted from the sample execution of this job on the smaller dataset [68]. This information is created for each platform of choice, e.g., *small, medium,* and *large* EC2 instances.
- *Job Scheduler:* it minimizes the overall completion time of a given set of MapReduce jobs by designing an optimized MapReduce jobs' execution. We apply the classic Johnson algorithm [35] described in Chapter 4.1.3 that was proposed as an optimal solution for two-stage production job schedule.
- *A simulator for Performance/Cost Analysis:* by varying the cluster size, the sim-

ulator generates the set of solutions across different platforms in a form of the trade-off curves: the cluster size and the achieved makespan define the *performance/cost)* metric for a given platform. These trade-off curves enable us to select the appropriate solution: the minimized makespan for a given cost or the minimized cost for achieving a given makespan.

JobProfiler

Our *Job Profiler* module uses the past job run(s) and extracts a *detailed job profile* that contains recorded durations of all map and reduce tasks. A similar job trace can be extracted from the Hadoop job tracker logs using tools such as Rumen [6]. The obtained map/reduce task distributions can be used for extracting the distribution parameters and generating scaled traces, i.e., generating the replayable traces of the job execution on the large dataset from the sample job execution on the smaller dataset as described in [68]. These job traces can be replayed using a MapReduce simulator [67] or used for creating the *compact job profile* for analytic models.

Optimized Job Schedule

As we showed in Chapter 4.1.3, the total completion time of n concurrent jobs depends significantly on the execution order of these jobs. For minimizing the makespan of a given set of MapReduce jobs $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ we apply the classic Johnson algorithm [35, 69] with more details described in Chapter 4.1.3.

The algorithm requires the knowledge of the map and reduce stage duration within a job. To estimate the stage durations within each job, we use a *compact job profile* that characterize the job execution during map, shuffle, and reduce phases via *average* and *maximum* task durations and then apply the MapReduce job model we introduced in Chapter 3.2.

Simulator

We use a event based Hadoop simulator called SimMR [67] that can replay execution traces of real workloads collected in Hadoop clusters. Figure 4.13 shows the overall design of SimMR.

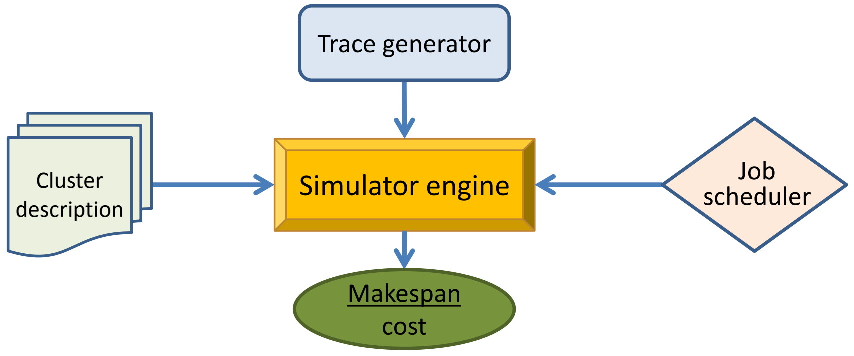


Figure 4.13: Simulator.

The basic blocks of the simulator are the following:

1. *Trace Generator* – a module that generates a replayable workload trace. This trace is generated either from the detailed job profile (provided by the Job Profiler) or by feeding the distribution parameters for generating the synthetic trace (this path is taken when we need to generate the job execution traces from the sampled executions on the smaller datasets).

2. *Simulator Engine* – a discrete event simulator that takes the cluster configuration information and accurately emulates the Hadoop job master decisions for map/reduce slot allocations across multiple jobs.

3. *Scheduling policy* – a scheduling module that dictates the jobs' ordering and the amount of allocated resources to different jobs over time.

Trace Generation We can generate job traces using two methods: JobProfiler and Synthetic TraceGen. JobProfiler extracts the job performance metrics by processing the counters and logs stored at the JobTracker at the end of each job. The job tracker logs reflect the MapReduce jobs' processing in the Hadoop cluster. They faithfully record the detailed information about the map and reduce tasks' processing. The logs also have useful information about the shuffle/sort stage of each job.

Alternatively, we can model the distributions of the durations based on the s-tatistical properties of the workloads and generate synthetic traces using Synthetic TraceGen. This can help evaluate hypothetical workloads and consider what-if s-cenarios. We store job traces persistently in a Trace database (for efficient lookup and storage) using a *job template*. The job template summarizes the job's essential performance characteristics during its execution in the cluster.

Simulator engine The simulator engine is the main component of SimMR which replays the given job trace. It manages all the discrete events in simulated time and performs the appropriate action on each event. It maintains data structures similar to the Hadoop job master such as a queue of submitted jobs $jobQ$. The slot allocation algorithm makes a new decision when a map or reduce task completes. Since our goal is to be fast and accurate, we simulate the jobs at the task level and do not simulate details of the TaskTrackers.

The simulator engine reads the job trace from the trace database. It communicates with the scheduler policies using a very narrow interface consisting of the following functions:

1. CHOOSENEXTMAPTASK(jobQ),
2. CHOOSENEXTREDUCETASK(jobQ)

These two functions ask the scheduling policy to return the jobId of the job whose map (or reduce) task should be executed next.

The simulator maintains a priority queue Q for *seven event types*: job arrivals and departures, map and reduce task arrivals and departures, and an event signaling the completion of the map stage. Each event is a triplet $(eventTime, eventType, jobId)$ where $eventTime$ is the time at which the event will occur in the simulation; $eventType$ is one of the seven event types; and $jobId$ is the job index of the job with which the event is associated.

The simulator engine fires events and runs the corresponding event handlers. It tracks the number of completed map and reduce tasks and the number of free slots. It allocates the map slots to tasks as dictated by the scheduling policy. When $minMapPercentCompleted$ percentage of map tasks are completed (it is the parameter set by the user), it starts scheduling reduce tasks. We could have started the reduce tasks directly after the map stage is complete. However, the shuffle phase of the reduce task occupies a reduce slot and has to be modeled as such. Hence, we schedule a filler reduce task of infinite duration and update its duration to the first shuffle duration when all the map tasks are complete. This enables accurate modeling of the shuffle phase.

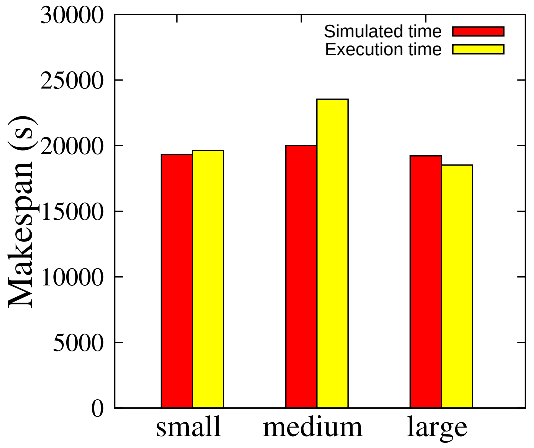
Scheduling policies Different scheduling and resource allocation policies can be used with SimMR for their evaluation, e.g., FIFO (Hadoop default scheduler), Fair [8], Capacity [2] and also our deadline-driven scheduler. While for the resource provision problem, we use the schedules that determined by Johnson's algorithm and the simulator engine will enforce the computed execution order of the jobs within the workload during the simulation.

Validation of Simulation Results

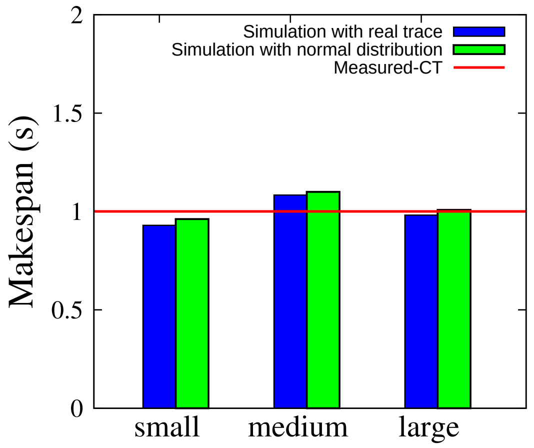
To validate the accuracy of the simulation results, we create a test workload $\mathcal{W}$ that contains 10 applications from Table 3.2:excludes the *Adjlist*, *KMeans* and *Classification*. We executed the workload on 3 Hadoop cluster formed with Amazon EC2 instance, each consists of the *small, medium,* and *large* EC2 instances respectively. In the experiments, we use 28 small instances, 20 medium instances and 24 large instances for each cluster. We choose these numbers because according to our simulation results, they all lead to a makespan of approximately 20000 seconds when execute the workload on each cluster (the execution order of jobs are determined by the Johnson's algorithm).

We then deploy the Hadoop clusters with the required number of instances and execute the workload $\mathcal{W}$ (with the corresponding Johnson job schedule) on the deployed clusters.

Figure 4.14 (a) shows the comparison between the simulated and the actual measured makespan (we repeated measurements 5 times). The results for *small* and *large* EC2 instances are very accurate. We can see a higher prediction error (17%) for *medium* instances. Partially, it is due to a higher variance in the job profile measurements collected on *medium* instances, especially during the shuffle phase.



(a) with different number of instances.



(b) with synthetic trace.

Figure 4.14: Simulator validation.

The simulation results in Figure 4.14 (a) are obtained by executing $\mathcal{W}$ in 3 clusters consists of the *small, medium* and *large* instances respectively and replaying the job traces collected in different platforms. Alternatively, the user can generate the synthetic traces defined by a given task duration distribution. This approach is especially attractive as it could also used to scale up trace, i.e., generate the replayable traces of the job execution on the large dataset from the sample job execution on the smaller dataset(s).

We generate synthetic job traces using a *normal distribution* with the mean μ and variance σ of the map/reduce task durations collected from the job sample executions on the 30-node Hadoop clusters with different instances. Figure 4.14 (b) shows the normalized makespan results that compare i) the simulated makespan with real (collected) job traces, ii) the simulated makespan with synthetically generated traces, and iii) the total completion time we measured during the actual execution of these jobs. In the testbed executions and the simulations, we use the same Johnson schedule within each cluster. As Figure 4.14 (b) shows both simulation with the recorded execution traces and the synthetically generated traces provide accurate makespan predictions (each one is within 10% of the real measurements).

4.2.2 Resource provision for homogeneous cluster

Based on the solution framework we proposed, we will first describe our resource provision strategy for deploying homogeneous clusters and then extend the approach to heterogeneous deployment which might brings us more benefits.

Figure 4.15 shows the diagram for the framework execution in decision making process per selected platform type. For example, if the platforms of interest

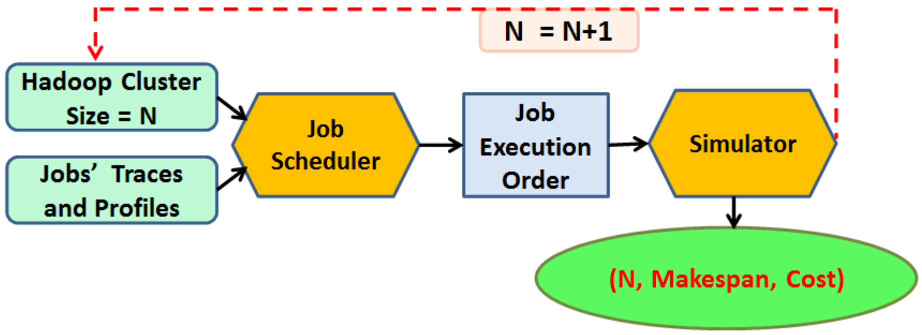


Figure 4.15: Solution Outline.

are *small*, *medium*, and *large* EC2 VM instances then the framework will generate three trade-off curves. For each platform and a given Hadoop cluster size, the *Job Scheduler* component generates the optimized MapReduce job schedule. Then the workload makespan is obtained by replaying the job traces in the simulator according to the generated schedule. After that the size of the cluster is increased by one instance (in the cloud environment, it is equivalent to adding a node to a Hadoop cluster) and the iteration is repeated: a new job schedule is generated and its makespan is evaluated with the simulator, etc. We have a choice of *stop* conditions for iterations: either a user can set a range of values for the cluster size (driven by the cost/budget constraints), or at some point, the increased cluster size does not improve the achievable makespan. The latter condition typically happens when the Hadoop cluster is large enough to accommodate all the jobs to be executed concurrently, and therefore, the increased cluster size cannot improve the jobs makespan.

Our solution is based on a simulation framework: in a brute-force manner, it searches through the entire solution space by exhaustively enumerating all possible candidates for the solution and checking whether each candidate satisfies the required problem's statement. Typically, the solution space is bounded by the budget $\mathcal{B}$, which a customer intends to spend. Assume that given jobs should be processed within deadline $\mathcal{D}$ and let $Price^{type}$ be the price of a *type* VM instance per time unit. Then the customer can rent N_{max}^{type} of VMs instances of a given *type*:

$$N_{max}^{type} = \mathcal{B}/(\mathcal{D} \cdot Price^{type}) \tag{4.5}$$

Algorithm 4 shows the pseudo code to determine the size of a cluster which is based on the *type* VM instances and which results in the minimal monetary cost.

The algorithm iterates through the increasing number of instances for a Hadoop cluster. It simulates the completion time of workload $\mathcal{W}$ processed with Johnson's schedule on a given size cluster and computes the corresponding cost (lines 2-6). Note, that k defines the number of *worker nodes* in the cluster. The overall Hadoop cluster size is $k+1$ nodes (we add a dedicated node for Job Tracker and Name Node, which is included in the *cost*). The min_cost^{type} keeps track of a minimal cost so far (lines 7-8) for a Hadoop cluster which can process $\mathcal{W}$ within deadline $\mathcal{D}$.

We apply Algorithm 4 to different types of VM instances, e.g., *small, medium*, and *large* respectively. After that we compare the produced outcomes and make a final provisioning decision.

Evaluation of resource provision for homogeneous cluster

We perform two case studies with two workloads $\mathcal{W}1$ and $\mathcal{W}2$ created from these applications:

Algorithm 4 Provisioning Solution for Homogeneous Cluster

```
Input:
$\mathcal{W} = \{J_1, J_2, ...J_n\} \leftarrow$ workload with traces and profiles for each job;
$type \leftarrow$ VM instance type, e.g., $type \in \{small, medium, large\}$;
$N^{type}_{max} \leftarrow$ the maximum number of instances to rent;
$Price^{type} \leftarrow$ unite price of a type VM instance;
$\mathcal{D} \leftarrow$ a given time deadline for processing $\mathcal{W}$.
Output:
$N^{type} \leftarrow$ an optimized number of VM type instances for a cluster;
$min\_cost^{type} \leftarrow$ the minimal monetary cost for processing $\mathcal{W}$.

1: $min\_cost^{type} \leftarrow \infty$
2: for $k \leftarrow 1$ to $N^{type}_{max}$ do
3:     // Simulate completion time for processing workload $\mathcal{W}$ with $k$ VMs
4:     $Cur\_CT = Simulate(type, k, \mathcal{W})$
5:     // Calculate the corresponding monetary cost
6:     $cost = Price^{type} \times (k+1) \times Cur\_CT$
7:     if $Cur\_CT \le \mathcal{D}$ & $cost < min\_cost^{type}$ then
8:         $min\_cost^{type} \leftarrow cost,\ N^{type} \leftarrow k$
9:     end if
10: end for
```

- $\mathcal{W}1$ – it contains all 13 applications shown in Table 3.2.
- $\mathcal{W}2$ – it contains ten applications: 1-8 and 10-11, i.e., excluding from the entire set the following three applications: *AdjList, Classification*, and *KMeans*.

We execute the set of 13 applications shown in Table 3.2 on three Hadoop clusters deployed with different types of EC2 VM instances (they can be obtained for the same price per time unit): *i)* 40 *small* VMs, *ii)* 20 *medium* VMs, and *iii)* 10 *large* VM instances. We configure these Hadoop clusters according to their nodes capacity as shown in Table 3.4, with 1 additional instance deployed as the NameNode and JobTracker.

These experiments pursue the following goals: i) to demonstrate the performance impact of executing these applications on the Hadoop clusters deployed with different EC2 instances; and 2) to collect the detailed job profiles for creating the job traces used for replay by the simulator and trade-off analysis in determining the optimal platform choice.

Figure 4.16 presents the completion times of 13 applications executed on the three different EC2-based clusters. The results show that the platform choice may significantly impact the application processing time. Note, we break the Y-axis as the *KMeans* and *Classification* applications take much longer time to finish compared to other applications. Figure 4.17 shows the normalized results with respect to the execution time of the same job on the Hadoop cluster formed with *small* VM instances. For 7 out of 13 applications, the Hadoop cluster formed with *small* instances leads to the best completion time (and the smallest cost). However, for the CPU-intensive applications such as *Classification* and *KMeans*, the Hadoop cluster formed with *large* instances shows better performance.

The presented results show that a platform choice for a Hadoop cluster may have a significant impact on the application performance. Moreover, the choice of the "right" platform becomes a challenging task if an objective is to minimize the makespan (the overall completion time) for a given budget or minimize the cost for achieving a given makespan.

Analyzing Performance and Cost Trade-Offs

Once the job profiles and job execution traces are collected, we can follow the proposed framework shown in Figure 4.15. For each platform of choice, i.e., *small, medium,* and *large* EC2 instances, and a given Hadoop cluster size, the *Job Scheduler* component generates the optimized MapReduce job schedule using the Johnson algorithm. After that the overall jobs' makespan is obtained by replaying the job traces in the simulator accordingly to the generated job schedule. At each iteration, the cluster size is increased by one instance (in this framework, it is equivalent to adding a node to a Hadoop cluster) and the process is repeated.

Workload $\mathcal{W}1$: Analysis of Performance and Cost Trade-Offs. Figure 4.18 (a) shows the simulation results for an achievable, optimized makespan of $\mathcal{W}1$ when it is processed by the Hadoop cluster with different number of nodes (instances) and the three platforms of choice: *small, medium,* and *large* EC2 instances.

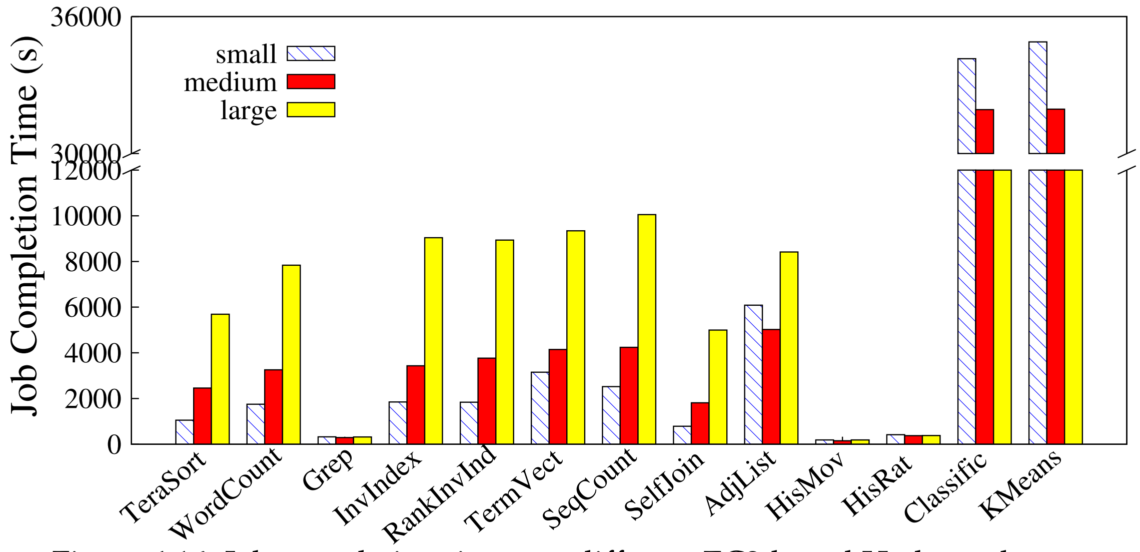


Figure 4.16: Job completion times on different EC2-based Hadoop clusters.

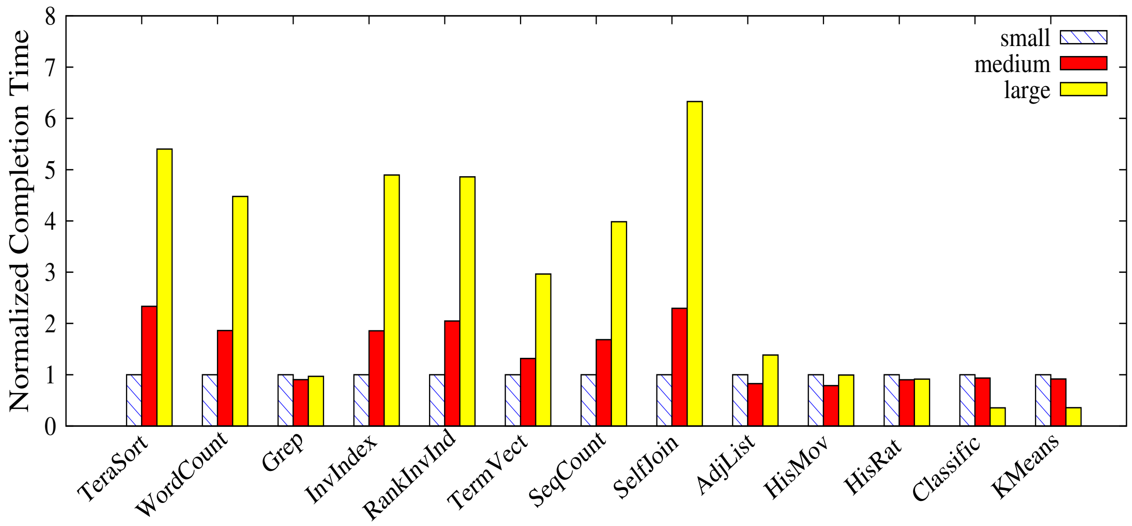
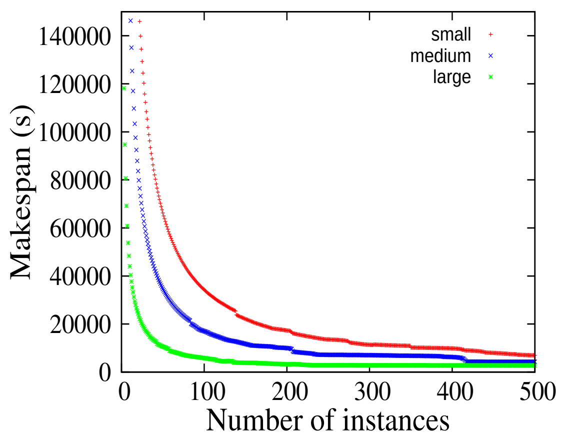


Figure 4.17: Normalized completion times on different EC2-based clusters.

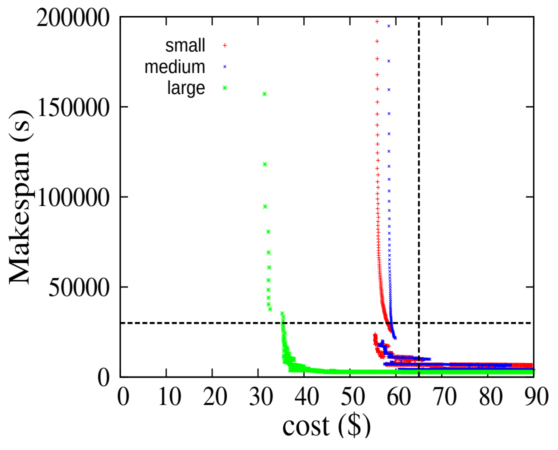
For each instance type, the jobs' makespan is inversely proportional to the Hadoop cluster size. However, there is a diminishing return for the cluster sizes above 100-200 nodes.

For each point along the curves in Figure 4.18 (a), we calculate the monetary cost for processing workload $\mathcal{W}1$. It is obtained by multiplying the makespan time, the corresponding cluster size, and the EC2 instance cost per unit of time (in seconds).

Figure 4.18 (b) shows the obtained performance and cost trade-offs for pro-



(a) Simulated makespan.



(b) Makespan vs. cost trade-offs.

Figure 4.18: Analysis of $\mathcal{W}1$ on the three platforms.

cessing workload $\mathcal{W}1$. The Y-axis shows the achievable makespan, while X-axis shows the corresponding cost across the three platforms of choice.

Each point along the curves in Figure 4.18 (b) corresponds to some point in Figure 4.18 (a). Note, there could be multiple points from Figure 4.18 (a) that result in either similar makespan or similar cost. At first, the makespan drops significantly with a small increase in the corresponding cost. It represents the case when the Hadoop cluster is small and adding more nodes could significantly reduce the total completion time. Therefore, while adding more nodes increases the cost, the improved makespan decreases the cluster time for "rent". The tail of the curve corresponds to the situation when the increased cluster size results in the increased cost but provides very little performance improvements to the achievable makespan.

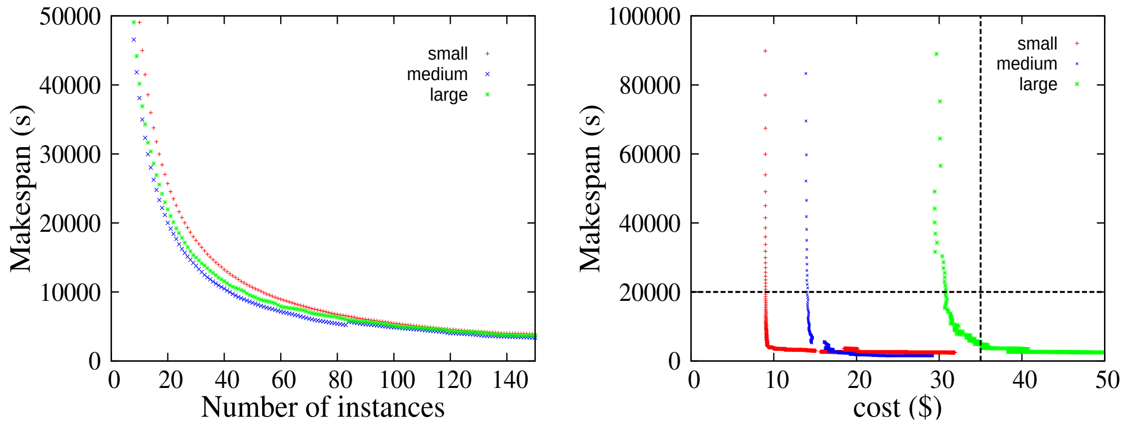
Another interesting observation is that not all the curve points in Figure 4.18 (b) are monotone (e.g., see the curves for the clusters with *small* and *medium* instances near the cost of \$60). The reason is that at some cluster sizes, adding one more node might reduce the number of waves (rounds) in the job execution and significantly improve the jobs' makespan. For example, we can clearly see in Figure 4.18 (a) such drop points around the cluster sizes of 200 nodes for the clusters with *small* and *medium* instances. As a result, the total cost drops significantly compared with the nearest points.

By searching along the generated trade-off curves in Figure 4.18 (b), we can determine the optimal solutions for the following problems: i) given a budget M, determine the platform type and the cluster size that leads to the minimized makespan; or ii) given a makespan target D, determine the platform type and the cluster size that minimizes the cost.

To demonstrate and quantify the benefits of our approach, let us select the makespan target of 30000 sec and the budget constraint of \$65. These constraints are shown in Figure 4.18 (b) with dashed lines. The minimal costs for processing workload $\mathcal{W}1$ with makespan target of 30000 seconds are \$55.47, \$56.38 and \$35.33 with *small, medium,* and *large* instances respectively. Therefore, by selecting the Hadoop cluster formed with *large* instances, we can save approx. 37% in monetary cost compared with clusters based on *medium* and *small* instances. Similarly, for the budget of \$65, the minimal makespan of 2805 seconds can be achieved by the Hadoop cluster formed with *large* instances. This results in 59.2% performance improvement compared with a makespan of 6874 seconds for Hadoop clusters based on *small* and *medium* instances.

Apparently, the Hadoop cluster formed with *large* EC2 instances is the best choice for processing $\mathcal{W}1$ workload for achieving different performance objectives.

Workload $\mathcal{W}2$: Analysis of Performance and Cost Trade-Offs. As demonstrated by Figures 4.19, the seven (out of ten) applications that comprise workload $\mathcal{W}2$ show a better (individual) performance on the Hadoop cluster formed with *small* instances. However, as in the case study with Workload $\mathcal{W}1$, it does not necessary imply that the Hadoop cluster formed with *small* EC2 instances will be the best cost/performance choice for meeting the performance objectives for executing the entire $\mathcal{W}2$ workload and minimizing its makespan. Figure 4.19 (a) shows the simulation results for an achievable, optimized makespan of $\mathcal{W}2$ when it is processed by the Hadoop cluster with different number of nodes (instances) and the three platforms of choice: *small*, *medium*, and *large* EC2 instances.



(a) Simulated makespan.　(b) Makespan vs. cost trade-offs.

Figure 4.19: Analysis of $\mathcal{W}2$ on the three platforms.

Now, using the results from Figure 4.19 (b), we calculate the makespan versus cost trade-offs curves for new workload $\mathcal{W}2$. These results are shown in Figure 4.19 (a).

To demonstrate and quantify the benefits of our approach, we select the makespan target of 20000 sec and the budget constraint of \$35 . These constraints are shown in Figure 4.19 (b) with dashed lines. Now, we can compare different achievable results for using Hadoop clusters formed with different types of EC2 instances. The minimal costs for processing workload $\mathcal{W}2$ with makespan target of 20000 seconds is \$9.0 provided by the *small* EC2 instances. This is 70.6% less compared to the case with *large* instances that can achieve the same makespan at a cost of \$30.67.

However, given a budget of \$35, the minimal makespan of 1487 seconds for processing workload $\mathcal{W}2$ can be achieved by the Hadoop cluster formed with *medium* instances. This leads to 69.4% performance improvement compared with a makespan of 4855 seconds for a Hadoop cluster based on the *large* instances. This outcome is interesting because it shows that the platform choice is non-trivial and additionally depends on the performance objectives.

The case studies with workloads $\mathcal{W}1$ and $\mathcal{W}2$ demonstrate that the choice of the

”right” platform for a Hadoop cluster is workload-dependent as well as influenced by the given performance objectives.

4.2.3 Resource provision for heterogeneous cluster

According to the example shown in Chapter 4.2, we analyzed the application performance of *TeraSort* and *KMeans* on different platforms, and observed that these applications benefit from different types of VMs as their preferred choice. Therefore, a single homogeneous cluster might not always be the best choice, and heterogeneous solution might offer a better cost/performance outcome.

As a motivating example, given a workload that contains two applications J_1, J_2, Figure 4.20 shows both the homogeneous provision and heterogeneous provision for completing the workload with the same time period. The homogeneous cluster (show in the left) contains 100 *small* instances while the heterogeneous cluster(show in the right) contains 50 *small* instances and 5 *large* instances. Suppose each small instance costs \$1 pre minute and each large instance costs \$4, then the homogeneous provision will cost $100 \times 1 \times 20 = \$2000$ in this case. However, the heterogeneous provision will cost $(50 \times 1 \times +5 \times 4) \times 20 = \$1400$ which saves \$600 compared with the homogeneous solution.

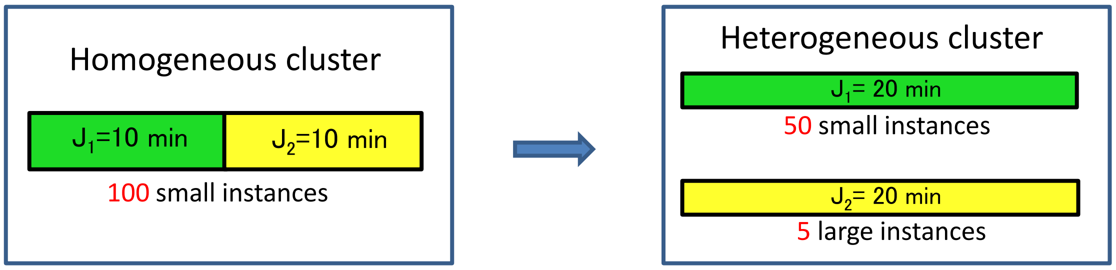
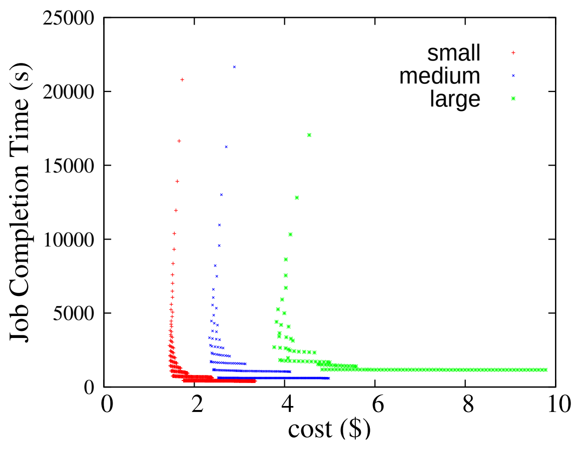


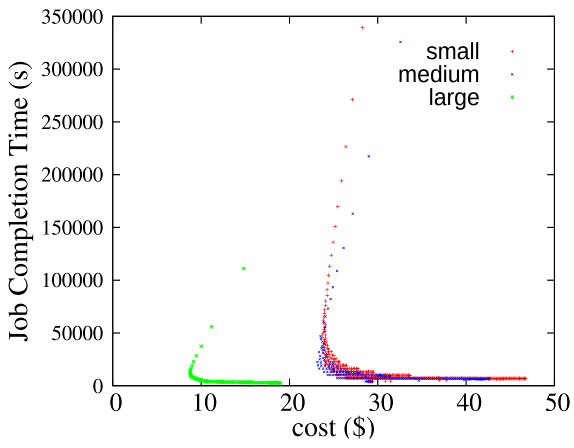
Figure 4.20: Possible benefits with heterogeneous cluster.

While each application has a preference platform (i.e. the platform that leads to better performance/costs) according to its characteristics, the preference choice of a give application often depends on the size of a Hadoop cluster and given performance goals. Continuing the motivating example from Chapter 4.2, Figure 4.21

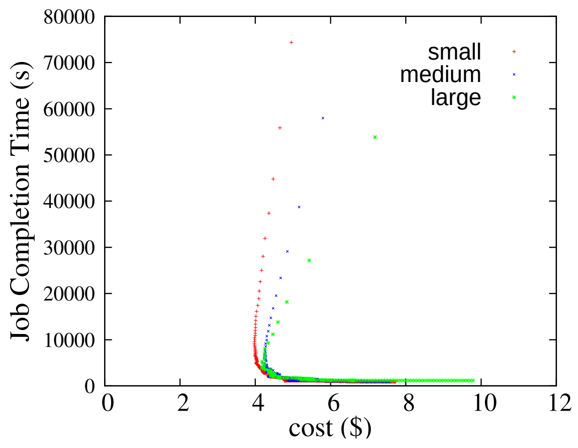
shows the trade-off curves for three representative applications *TeraSort*, *KMeans*, and *AdjList* obtained as a result of exhaustive simulation of application completion times on different size Hadoop clusters. The Y-axis represents the job completion time while the X-axis shows the corresponding monetary cost. Each figure shows three curves for application processing by a homogeneous Hadoop cluster based on *small*, *medium*, and *large* VM instances respectively.



(a) *TeraSort*



(b) *KMeans*



(c) *AdjList*

Figure 4.21: Performance versus cost trade-offs for different applications.

First of all, the same application can result in different completion times when being processed on the same platform at the same cost. This reflects an interesting phenomenon of "pay-per-use" model. There are situations when a cluster of size N processes a job in T time units, while a cluster of size $2 \cdot N$ may process the same job in $T/2$ time units. Interestingly, these two different size clusters have the same cost, and if the purpose is meeting deadline $\mathcal{D}$ where $T \leq \mathcal{D}$ then both clusters meet the performance objective.

Second, we can see an orthogonal observation: in many cases, the same completion time can be achieved at a different cost (on the same platform type). Typically, this corresponds to the case when an increased size Hadoop cluster does not further improve the job processing time.

Finally, according to Figure 4.21, we can see that for *TeraSort*, the *small* instances results in the best choice, while for *KMeans* the *large* instances represent the most cost-efficient platform. However, the optimal choice for *AdjList* is not very clear, it depends on the deadline requirements, and the trade-off curves are much closer to each other than for *TeraSort* and *KMeans*.

Another important point is that the cost savings vary across different applications, e.g., the execution of *KMeans* on *large* VM instances leads to higher cost savings than the execution of *TeraSort* on *small* VMs. Thus, if we would like to partition a given workload $\mathcal{W} = \{J_1, J_2, ...J_n\}$ into two groups of applications each to be executed by a Hadoop cluster based on different type VM instances, we need to be able to **rank** these application with respect to their preference "strength" between two considered platforms.

In the next section, we consider a heterogeneous solution that consists of two homogeneous Hadoop sub-clusters deployed with different type VM instances[2]. As an example, we consider a heterogeneous solution formed by *small* (S) and *large* (L) VM instances. To measure the "strength" of application preference between two different VM types we introduce an application **preference score** $PScore^{S-L}$ defined as a difference between the normalized costs of simulated cost/performance curves (such as shown in Figure 4.21):

$$PScore^{S-L} = \frac{\sum_{1 \leq i \leq N^S_{max}} Cost^S_i}{N^S_{max}} - \frac{\sum_{1 \leq i \leq N^L_{max}} Cost^L_i}{N^L_{max}} \tag{4.6}$$

where N^S_{max} and N^L_{max} are defined by Eq. 4.5 for Hadoop clusters with *small* and *large* VM type instances respectively.

[2]The designed framework can be generalized for a larger number of clusters. However, this might significantly increase the algorithm complexity without adding new performance benefits.

The value of $PScore^{S-L}$ indicates the possible impact on the provisioning cost, i.e, a large negative (positive) value indicates a stronger preference of *small* (*large*) VM instances, while values closer to 0 reflect less sensitivity to the platform choice.

For optimized heterogeneous solution, we need to determine the following parameters:

- The number of instances for each sub-cluster (i.e., the number of worker nodes plus a dedicated node to host JobTracker and Name Node for each sub-cluster).
- The subset of applications to be executed on each cluster.

Algorithm 5 shows the pseudo code of our heterogeneous solution. For a presentation simplicity, we show the code for a heterogeneous solution with *small* and *large* VM instances.

First, we sort the jobs in the ascending order according to their preference ranking $PScore^{S-L}$. Thus the jobs in the beginning of the list have a performance preference for executing on the *small* instances. Then we split the ordered job list into two subsets: first one to be executed on the cluster with *small* instances and the other one to be executed on the cluster with *l*arge instances (lines 4-5). For each group, we use Algorithm 4 for homogeneous cluster provisioning to determine the optimized size of each sub-cluster that leads to the minimal monetary cost (lines 6-7). We consider all possible splits by iterating through the split point from 1 to the total number of jobs N and use a variable min_cost^{S+L} to keep track of the found minimal total cost, i.e, the sum of costs from both sub-clusters (lines 9-12).

Evaluation of resource provision for heterogeneous cluster

In the performance study, we use the same set of 13 applications described in Section 3.2. Table 4.2 provides the description of the applications with the application preference score $PScore^{S-L}$ in the last column

A positive value of the application preference score $PScore^{S-L}$ (e.g, *KMeans, Classification*) indicates that the application is more cost-efficient on *large* VMs,

Algorithm 5 Provisioning Solution for Heterogeneous Cluster

Input:
$\mathcal{W} = \{J_1, J_2, ...J_n\} \leftarrow$ workload with traces and profiles, where jobs are sorted in ascending order by their preference score $PScore^{S-L}$;
$\mathcal{D} \leftarrow$ a given time deadline for processing $\mathcal{W}$.
Output:
$N^S \leftarrow$ number of *small* instances;
$N^L \leftarrow$ number of *large* instances;
$\mathcal{W}^S \leftarrow$ List of jobs to be executed on *small* instance-based cluster;
$\mathcal{W}^L \leftarrow$ List of jobs to be executed on *large* instance-based cluster;
$min\_cost^{S+L} \leftarrow$ the minimal monetary cost of heterogeneous clusters.

```
1: $min\_cost^{S+L} \leftarrow \infty$
2: for $split \leftarrow 1$ to $n-1$ do
3:     // Partition workload $\mathcal{W}$ into 2 groups
4:     $Jobs^S \leftarrow J_1, ..., J_{split}$
5:     $Jobs^L \leftarrow J_{split+1}, ..., J_n$
6:     $(\tilde{N^S}, min\_cost^S)$ = Algorithm 4$(Jobs^S, small, \mathcal{D})$
7:     $(\tilde{N^L}, min\_cost_L)$ = Algorithm 4$(Jobs^L, large, \mathcal{D})$
8:     $total\_cost \leftarrow min\_cost^S + min\_cost^L$
9:     if $total\_cost < min\_cost^{S+L}$ then
10:        $min\_cost^{S+L} \leftarrow total\_cost$
11:        $\mathcal{W}^S \leftarrow Jobs^S$,  $\mathcal{W}^L \leftarrow Jobs^L$
12:        $N^S \leftarrow \tilde{N^S}$,  $N^L \leftarrow \tilde{N^L}$
13:    end if
14: end for
```

while a negative value (e.g., *TeraSort*, *WordCount*) means that the application favors *small* VM instances. The absolute score value is indicative of the preference "strength". When the preference score is close to 0 (e.g., *Adjlist*), it means that the application does not have a clear preference between the instance types.

We create *three* different workloads described as follows:

- $\mathcal{W}1$ – it contains all 13 applications shown in Table 3.2.
- $\mathcal{W}2$ – it contains 11 applications: 1-11, i.e., excluding *KMeans* and *Classification* from the application set.
- $\mathcal{W}3$ – it contains 12 applications: 1-12, i.e., excluding *KMeans* from the application set.

Figure 4.22 shows the simulated cost/performance trade-off curves for three workloads executed on both homogeneous and heterogeneous Hadoop cluster(s).

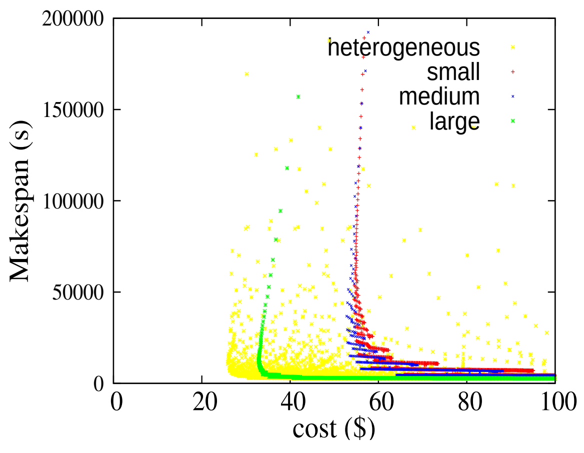
Application	*Input data (type)*	*Input data size (GB)*	*#map,red tasks*	$PScore^{S-L}$
1. *TeraSort*	Synthetic	31	495, 240	-3.74
2. *WordCount*	Wikipedia	50	788, 240	-5.96
3. *Grep*	Wikipedia	50	788, 1	-3.30
4. *InvIndex*	Wikipedia	50	788, 240	-7.90
5. *RankInvIndex*	Wikipedia	46	745, 240	-5.13
6. *TermVector*	Wikipedia	50	788, 240	3.11
7. *SeqCount*	Wikipedia	50	788, 240	-4.23
8. *SelfJoin*	Synthetic	28	448, 240	-5.41
9. *AdjList*	Synthetic	28	508, 240	-0.7
10. *HistMovies*	Netflix	27	428, 1	-1.64
11. *HistRatings*	Netflix	27	428, 1	-2.53
12. *Classification*	Netflix	27	428, 50	19.59
13. *KMeans*	Netflix	27	428, 50	18.6

Table 4.2: Application characteristics.

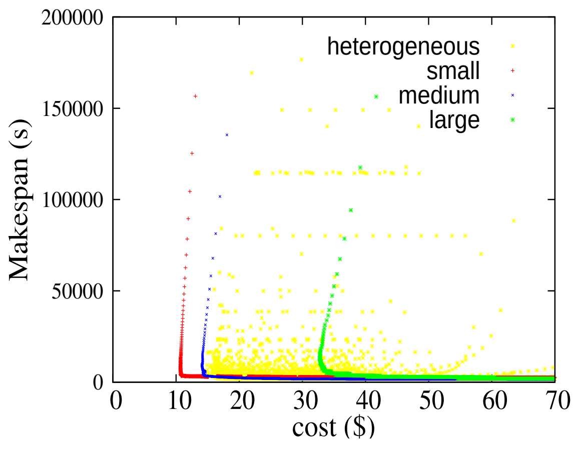
For *homogeneous provisioning*, we show the three trade-off curves for Hadoop clusters based on *small, medium* and *large* VM instances respectively.

Figure 4.22 (a) shows that workload $\mathcal{W}1$ is more cost-efficient when executed on the Hadoop cluster with *large* VMs (among the homogeneous clusters). Such results can be expected because $\mathcal{W}1$ contains both *KMeans* and *Classification* that have very strong preference towards *large* VM instances (see their high positive $PScore^{S-L}$). In comparison, $\mathcal{W}2$ contains applications that mostly favor the *s-mall* VM instances, and as a result, the most efficient trade-off curve belongs to a Hadoop cluster based on the *small* VM instances. Finally, $\mathcal{W}3$ represents a mixed case: it has *Classification* application that strongly favors *large* VM instances while most of the remaining applications prefer *small* VM instances. Figure 4.22(c) shows that a choice of the best homogeneous platform depends on the workload performance objectives (i.e., deadline $\mathcal{D}$).

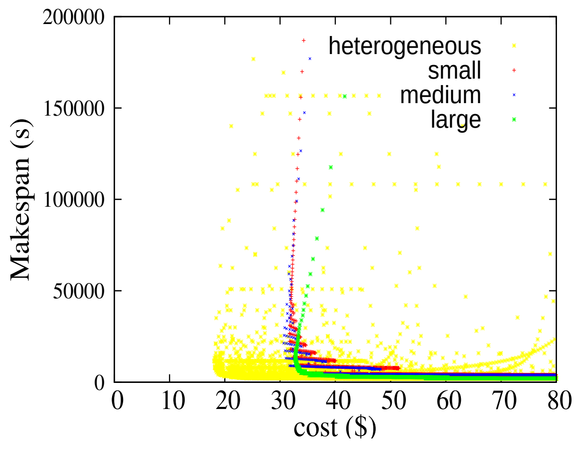
The yellow dots in Figure 4.22 represent the completion time and monetary cost when we exploit a *heterogeneous provisioning* case. Each point corresponds to a workload split into two subsets that are executed on the Hadoop cluster formed with *small* and *large* VM instances respectively. This is why instead of the explicit



(a) Workload $\mathcal{W}1$



(b) Workload $\mathcal{W}2$



(c) Workload $\mathcal{W}3$

Figure 4.22: Performance versus cost trade-offs for different workloads.

trade-off curves for the homogeneous cluster case, the simulation results for the heterogeneous case look much more scattered across the space.

To evaluate the efficiency of our provisioning algorithms, we consider different performance objectives for each workload:

- $\mathcal{D}$= 20000 seconds for workload $\mathcal{W}1$;
- $\mathcal{D}$= 10000 seconds for workload $\mathcal{W}2$;
- $\mathcal{D}$= 15000 seconds for workload $\mathcal{W}3$.

Tables 4.3-4.5 present the provisioning results for each workload with homogeneous and heterogeneous Hadoop clusters that have minimal monetary costs while meeting the given workload deadlines.

Among the homogeneous Hadoop clusters for $\mathcal{W}1$, the cluster with *large* VM instances has the lowest monetary cost of \$32.86.

By contrast, for workload $\mathcal{W}2$, the homogeneous Hadoop cluster with *small* VMs provides the lowest cost of \$10.68.

For $\mathcal{W}3$, all the three homogeneous solutions lead to a similar minimal cost, and the Hadoop cluster based on *medium* VMs has a slightly better cost than the other two alternatives.

Intuitively, these performance results are expected from the trade-off curves for three workloads shown in Figure 4.22.

Cluster Type	Number of Instances	Completion Time (sec)	Monetary Cost (\$)
small (homogeneous)	210	15763	55.43
medium (homogeneous)	105	15137	53.48
large (homogeneous)	39	12323	32.86
small + *large* heterogeneous	48 *small* + 20 *large*	14988	**24.21**

Table 4.3: Cluster provisioning results for workload $\mathcal{W}1$.

Cluster type	Number of Instances	Completion Time (sec)	Monetary Cost (\$)
small (homogeneous)	87	7283	**10.68**
medium (homogeneous)	43	9603	14.08
large (homogeneous)	49	9893	32.98
small+ *large* heterogeneous	76 *small* + 21 *large*	6763	14.71

Table 4.4: Cluster provisioning results for workload $\mathcal{W}2$.

Cluster type	Number of Instances	Completion Time (sec)	Monetary Cost (\$)
small (homogeneous)	140	13775	32.37
medium (homogeneous)	70	13118	31.05
large (homogeneous)	36	13265	32.72
small + *large* heterogeneous	74 *small* + 15 *large*	10130	**18.0**

Table 4.5: Cluster provisioning results for workload $\mathcal{W}3$.

The best *heterogeneous solution* for each workload is shown in the last row in Tables 4.3-4.5. For $\mathcal{W}1$, the minimal cost of the heterogeneous solution is \$24.21 which is **26%** improvement compared to the minimal cost of the homogeneous solution based on the *large* VM instances. In this heterogeneous solution, the applications *SelfJoin, WordCount, InvIndex* are executed on the cluster with *small* VMs and applications *Classification, KMeans, TermVector, Adjlist, HistMovies, HistRating, Grep, TeraSort, SeqCount, RankInvInd* are executed on the cluster with *large* VM instances.

The cost benefits of the heterogeneous solution is even more significant for $\mathcal{W}3$: it leads to cost savings of **42%** compared to the minimal cost of the homogeneous solution. In this heterogeneous solution, the applications *HistMovies, HistRating, Grep, TeraSort, SeqCount, RankInvInd, SelfJoin, WordCount, InvIndex* are executed on the cluster with *small* VMs and applications *Classification, TermVector, Adjlist* are executed on the cluster with *large* VMs.

However, for workload $\mathcal{W}2$, the heterogeneous solution does not provide additional cost benefits. One important reason is that for a heterogeneous solution, we need to maintain additional nodes deployed as JobTracker and NameNode for each sub-cluster. This increases the total provisioning cost compared to the homogeneous solution which only requires a single additional node for the entire cluster. The workload properties also play an important role here. As $\mathcal{W}2$ workload does not have any applications that have "strong" preference for *large* VM instances, the introduction of a special sub-cluster with *large* VM instances is not justified.

4.3 Conclusion

In this chapter, we focus on two resource management problems for MapReduce workloads. One is resource allocation on shared Hadoop cluster which aims to tailer and control the amount of cluster resources that should be allocated to each application. The other is resource provisioning in public cloud environment which aims to help users to select the the best cost/performance platform for a given workload that contains multiple applications with different platform preferences.

We solve these problems by combining the profiling strategy, the platform evaluation framework described in Chapter 3 and also a simulator that replays the extracted traces with different platform settings. Specifically, for the resource allocation problem, we use our evaluation framework to accurately estimate the amount of resource that required by each application and our deadline-driven scheduler controls the resource allocation during runtime. For the resource provisioning problem, we demonstrate that seemingly equivalent platform choices for a Hadoop cluster might result in a very different application performance, and thus lead to a different cost. We propose our solution for both homogeneous and heterogeneous clusters. Our case study with Amazon EC2 platform reveals that for different workloads, an optimized platform choice may result in 41%-67% cost savings for achieving the same performance objectives. Moreover, depending on the workload characteristic, the heterogeneous solution may outperform the homogeneous cluster solution by 26%-42%.

Chapter 5

Performance Optimization with optimal job settings

How to execute the MapReduce applications more efficiently is an important issue which has been studied in several works [46, 71, 34, 75]. In this chapter, we focus on optimizing the execution performance of MapReduce applications that defined as a workflow of *sequential* jobs through automatically turning the job settings along the workflow.

We first show in Chapter 5.1 that the number of reduce task could significantly affect the MapReduce job completion time and the choice of the right number of reduce tasks depends on the Hadoop cluster size, the size of the input dataset(s) of the job, and the amount of resources available for processing this job. For more complex applications defined as MapReduce workflows, the effect of the job settings could also propagate through the workflow due to the data dependency: the output of one job becomes the input of the next job, and therefore, the number of reduce tasks in the previous job defines the number (and size) of input files of the next job, and affect its processing efficiency. Besides, we also identify the performance trade-offs during the application execution: depend on the application property and the input data size, a nearly optimal completion time might be achieved with a relatively small amount of the cluster resources.

We then provide an automatic performance optimization tool that automates the user efforts of tuning the job settings within a MapReduce application. i.e., tuning the numbers of reduce tasks along the MapReduce workflow. It is based on the performance modeling framework we proposed in Chapter 3 to evaluate the execution efficiency of different job settings and aims to determine the one for optimizing the total completion time while minimizing the resource usage for its execution. It adopts two optimization strategies to achieve trade-offs between these two goals: a local one with trade-offs at a job level, and a global one that makes the optimization trade-off decisions at the workflow level.

5.1 Motivation

Figure 5.1 shows a simple motivating example for the problem of tuning the job settings and its impact on the completion time. In these experiments, we use the *Sort benchmark* [47] with 10 GB input on 64 machines each configured with a single map and a single reduce slot, i.e., with 128 map and 128 reduce slots overall.

Figure 5.1 (a) shows the job completion time as different numbers of reduce tasks are used for executing this job. The configurations with 64 and 128 reduce tasks produce much better completion times compared with other settings shown in this graph (10%-45% completion time reduction). Intuitively, settings with a low number of reduce tasks limit the job execution concurrency. While, settings with a higher number of reduce tasks increase the job execution parallelism but they also require a higher amount of resources (slots) assigned to the program. Moreover, at some point (e.g., 512 reduce tasks) it may lead to a higher overhead and higher processing time.

Figure 5.1 (b) shows a complementary situation: it reflects how the job completion time is impacted by the input data size per map task. In a MapReduce workflow, the outputs generated by the previous job become inputs of the next one, and the size of the generated files may have a significant impact of the performance of the next job. In these experiments, we use the same *Sort benchmark* [47] with 10 GB input, which has a fixed number of 120 reduce tasks, but the input

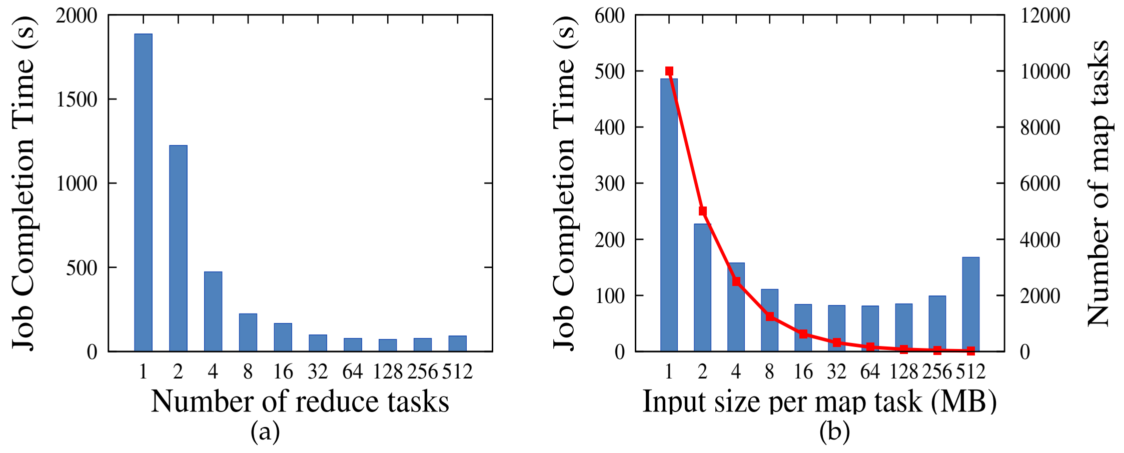


Figure 5.1: Motivating Examples.

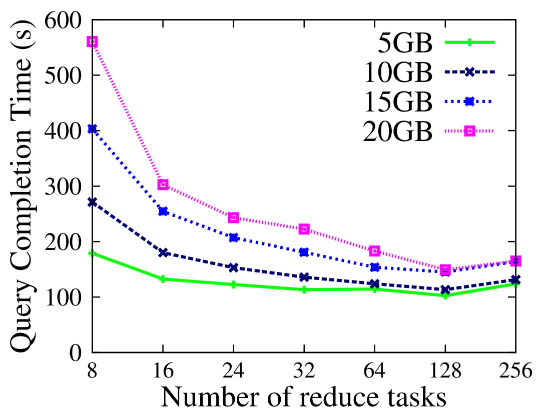
file sizes of map tasks are different. The line that goes across the bars reflects the number of map tasks executed by the program (practically, it shows the concurrency degree in the map stage execution). The interesting observation here is that the smaller size input per task incur higher processing overhead that overwrites the benefits of a high execution parallelism. However, a larger input size per map task limits the concurrency degree in the program execution. Another interesting observation in Figure 5.1 (b) is that there are a few different input sizes per map task that result in a similar completion time, but differ in how many map slots are needed for job processing.

5.1.1 Why not use best practices?

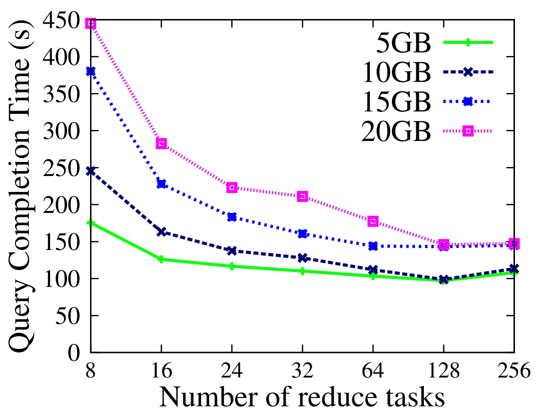
There is a list of best practices [3] that offers useful guidelines to the users in determining the appropriate configuration settings. The offered *rule of thumb* suggests to set the number of reduce tasks to 90% of all available resources (reduce slots) in the cluster. Intuitively, this maximizes the concurrency degree in job executions while leaving some "room" for recovering from the failures. This approach may work under the FIFO scheduler when all the cluster resource are (eventually) available to the next scheduled job. This guideline does not work well when the Hadoop cluster is shared by multiple users, and their jobs are scheduled with

Hadoop Fair Scheduler (HFS) [77] or Capacity Scheduler [2]. Moreover, the rule of thumb suggests the same number of reduce tasks for all applications without taking into account the amount of input data for processing in these jobs.

To illustrate these issues, Figure 5.2 shows the impact of the number of reduce tasks on the query completion time for *TPC-H Q1* and *TPC-H Q19* with different input dataset sizes. Both queries are compiled into workflows with two sequential MapReduce jobs (See the query description in Chapter 5.3.1).



(a) *TPC-H Q1*



(b) *TPC-H Q19*

Figure 5.2: Effect of reduce task settings for processing the same job with different input dataset sizes.

The *rule of thumb* suggests to use 115 reduce tasks (128*0.9=115). However, as we can see from the results in Figure 5.2 (a), for dataset sizes of 10 GB and 15 GB the same performance could be achieved with 50% of the suggested resources. The resource savings are even higher for *TPC-H Q1* with 5 GB input size: it can achieve the nearly optimal performance by using only 24 reduce tasks (this represents 80% savings against the *rule of thumb* setting). The results for *TPC-H Q19* show similar trends and conclusion.

In addition, Figure 5.3 shows the effect of reduce task settings on *TPC-H Q1* query completion time when only a fraction of resources (both map and reduce slots) is available for the job execution.

Figures 5.3 (a) and (b) show the results with the input dataset size of 10 GB and 1 GB respectively. The graphs reflect that when less resources are available to a job

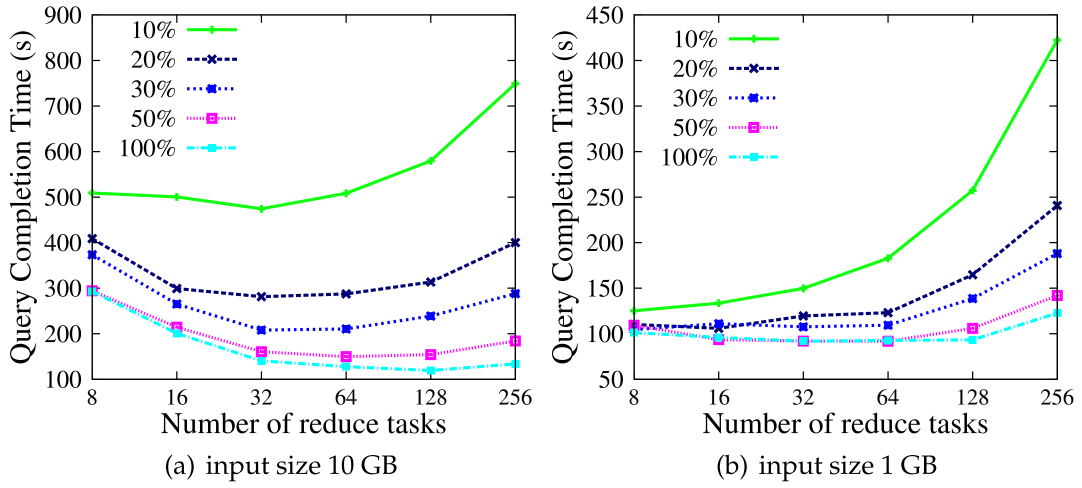


Figure 5.3: Effect of reduce task settings when only a fraction of resources is available.

(e.g., 10% of all map and reduce slots in the cluster), the offered *rule of thumb* setting (115 reduce tasks) could even hurt the query completion time since the expected high concurrency degree in the job execution cannot be achieved with limited resources while the overhead introduced by a higher number of reduce tasks causes a longer completion time. This negative impact is even more pronounced when the input dataset size is small as shown in Figure 5.3 (b). For example, when the query can only use 10% of cluster resources, the query completion time with the rule of thumb setting (115 reduce tasks) is more than 2 times higher compared to the completion time of this query with eight reduce tasks.

5.2 Problem definition and the solution outline

Our *main goal* is to determine the job settings (i.e., the number of reduce tasks for each job) that optimize the overall completion time of the application. However, we also aim for an *additional goal* that is to minimize the resource usage for achieving this optimized time. Often nearly optimal completion time can be achieved with a significantly smaller amount of resources (see the outcome of 64 and 128 reduce tasks in Figure 5.1 (a)). Moreover, it was observed in [77, 73, 61, 60] that the lack of reduce slots in the Hadoop cluster is a main cause of a starvation problem. Therefore, optimizing (decreasing) the reduce task settings while achieving performance objectives is a desirable feature of an efficient workload management in the cluster.

Towards solving the problem, one critical observation we have is that the optimization problem for complex MapReduce workflows can be efficiently solved through the optimization problem of the pairs of its sequential jobs.

As an example, Figure 5.4 shows a MapReduce workflow that consists of three sequential jobs: J_1, J_2, and J_3. To optimize the overall completion time we need to

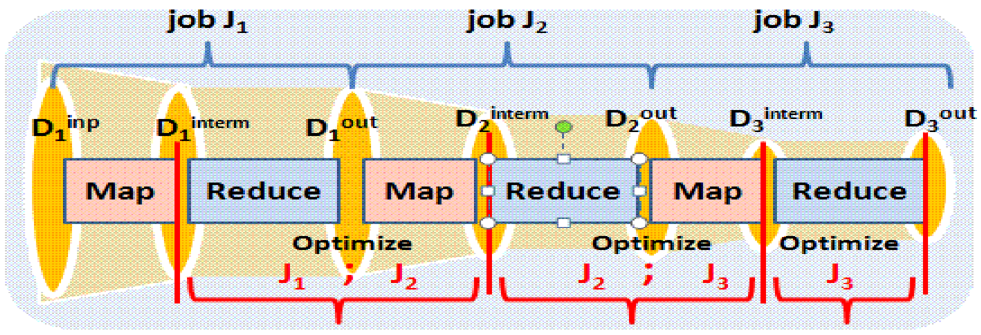


Figure 5.4: Example workflow with 3 sequential jobs

tune the reduce task settings in jobs J_1, J_2, and J_3. A question to answer is whether the choice of reduce task setting in job J_1 impacts the choice of reduce task setting in job J_2, etc.

A critical observation here is that *the size of overall data* generated between the map and reduce stages of the same job and between two sequential jobs *does not depend on the reduce task settings* of these jobs. For example, the overall amount of output data D_1^{out} of job J_1 does not depend on the number of reduce tasks in J_1. It is defined by the size and properties of D_1^{interm}, and the semantics of J_1's reduce function. Similarly, the amount of D_2^{interm} is defined by the size of D_1^{out}, properties of this data, and the semantics of J_2's map function. Again, the size of D_2^{interm} does not depend on the number of reduce tasks in J_1.

Therefore the amount of intermediate data generated by the map stage of J_2 is the same (i.e., *invariant*) for different settings of reduce tasks in the previous job J_1.

It means that the choice of an appropriate number of reduce tasks in job J_2 does not depend on the choice of reduce task setting of job J_1. It is primarily driven by an optimized execution of the next pair of jobs J_2 and J_3. Finally, tuning the reduce task setting in J_3 is driven by optimizing its own completion time.

In such a way, the optimization problem of the entire workflow can be efficiently solved through the *optimization problem of the pairs of its sequential jobs*. Therefore, for two sequential jobs J_1 and J_2, we need to design a model that evaluates the execution times of J_1's reduce stage and J_2's map stage as a function of a number of reduce tasks in J_1. Such a model will enable us to iterate through a range of reduce tasks' parameters and identify a parameter that leads to the minimized completion time of these jobs.

5.2.1 Two optimization strategies

According to the observation we described in Chapter 5.2, the optimization problem of reduce task settings for a given MapReduce workflow $W = \{J_1, ..., J_n\}$ can be efficiently solved through the *o*ptimization problem of the pairs of its sequential jobs. Therefore, for any two sequential jobs (J_i, J_{i+1}), where $i = 1, ..., n-1$, we need to evaluate the execution times of J_i's reduce stage and J_{i+1}'s map stage as a function of the number of reduce tasks $N_R^{J_i}$ in J_i (see the related illustration in Figure 5.4). Let us denote this execution time as $T_{i,i+1}(N_R^{J_i})$.

By iterating through the number of reduce tasks in J_i we can find the reduce task setting $N_R^{J_i,min}$ that results in the minimal completion time $T_{i,i+1}^{min}$ for the pair (J_i, J_{i+1}), i.e., $T_{i,i+1}^{min} = T_{i,i+1}(N_R^{J_i,min})$. By determining the reduce task settings s for all the job pairs, i.e., $s^{min} = \{N_R^{J_1,min}, ..., N_R^{J_n,min}\}$, we can determine the minimal total completion time $T_W(s^{min})$. The optimizations can be used with Hadoop Fair Scheduler (HFS) [8] or Capacity Scheduler [2] and multiple jobs executed on a cluster. Both schedulers allow configuring different size resource pools each running jobs in the FIFO manner.

Note, that the proposed approach for finding the reduce task setting that minimizes the total completion time can be applied to a different amount of available resources, e.g., the entire cluster or a fraction of available cluster resources. In such a way, the optimized execution can be constructed for any size resource pool managed (available) in a Hadoop cluster.

We aim to design the optimization strategy that enables a user to analyze the possible trade-offs, such as execution performance versus its resource usage. We aim to answer the following question: if the performance goal allows a specified increase of the minimal total completion time $T_W(s^{min})$, e.g., by 10%, then what is the resource usage under such execution compared to $R_W(s^{min})$?

We define the resource usage $R_{i,i+1}(N_R^{J_i})$ for a sequential job pair (J_i, J_{i+1}) executed with the number of reduce tasks $N_R^{J_i}$ in job J_i as follows:

$$R_{i,i+1}(N_R^{J_i}) = T_{R\_task}^{J_i} \times N_R^{J_i} + T_{M\_task}^{J_{i+1}} \times N_M^{J_{i+1}}$$

where $N_M^{J_{i+1}}$ represent the number of map tasks of job J_{i+1}, and $T_{R\_task}^{J_i}$ and $T_{M\_task}^{J_{i+1}}$ represent the average execution time of reduce and map tasks of J_i and J_{i+1} respectively. The resource usage for the entire MapReduce workflow is defined as the sum of resource usages for each job within the workflow.

Table 5.1 summarizes the notations that we use for defining the optimization strategies below.

Table 5.1: Notation Summary

$T_{i,i+1}(N_R^{J_i})$	Completion time of (J_i, J_{i+1}) with $N_R^{J_i}$ reduce tasks
$R_{i,i+1}(N_R^{J_i})$	Resource usage of pair (J_i, J_{i+1}) with $N_R^{J_i}$ reduce tasks
$T_W(s)$	Completion time of the entire workflow W with setting s
$R_W(s)$	Resource usage of the entire workflow W with setting s
$T_{i,i+1}^{min}$	Minimal completion time of a job pair (J_i, J_{i+1})
$N_R^{J_i,min}$	Number of reduce tasks in J_i that leads to $T_{i,i+1}^{min}$
$w_$increase	Allowed increase of the min workflow completion time
$N_R^{J_i,incr}$	Number of reduce tasks in J_i to meet the increased time

The first algorithm is based on the *local optimization*. The user specifies the allowed increase *w_increase* of the minimal overall completion time $T_W(s^{min})$. Our goal is to compute the new job settings that allow achieving this increased completion time.

To accomplish this goal, a straightforward approach is to apply the user-defined *w_increase* to the minimal completion time $T_{i,i+1}^{min}$ of each pair of sequential jobs (J_i, J_{i+1}), and then determine the corresponding number of reduce tasks in J_i. Figure 5.5 illustrates the possible relationship between the completion time of job pair (J_i, J_{i+1}) and the number of reduce tasks in j_i. The horizontal red line represents the relaxed completion time when applied the user defined threshold on the minimal pair duration and the vertical green line is the corresponding setting of the reduce tasks in satisfying the new completion time. We could find out such new setting by step-by-step decrease the reduce task number (start from the one that leads to the minimal duration) and recompute the pair completion time.

The pseudo-code defining this strategy is shown in Algorithm 6. The completion time of each job pair is locally increased (line 2), and then the corresponding reduce task settings are computed (lines 4-6).

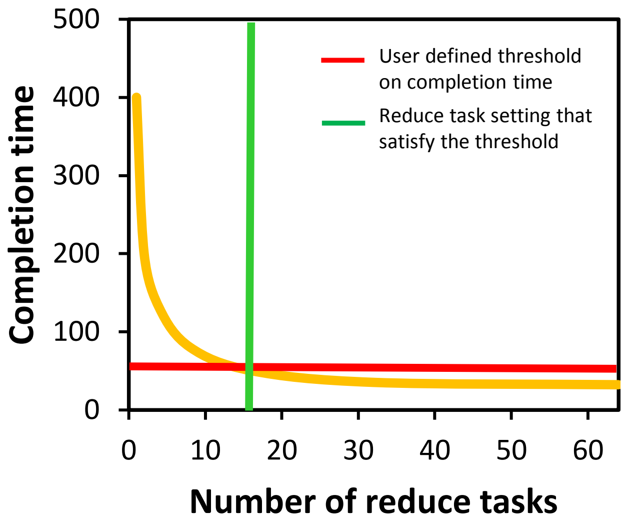


Figure 5.5: Example of the local optimization strategy

While this local optimization strategy is simple to implement, there could be additional resource savings achieved if we consider a *global optimization*. Intuitively, the resource usage for job pairs along the workflow might be quite different depending on the job characteristics. Therefore, we could identify the job pairs with the highest resource savings (gains) for their increased completion times.

The pseudo-code defining this global optimization strategy is shown in Algo-

Algorithm 6 Local optimization strategy for deriving workflow reduce tasks' settings

```
1: for $i \leftarrow 1$ to $n$ do
2:     $T_{i,i+1}^{incr} = T_{i,i+1}^{min} \times (1 + w\_increase)$
3:     $N_R^{J_i,cur} \leftarrow N_R^{J_i,min}$
4:     while $T_{i,i+1}(N_{R_i}^{J_i,cur}) < T_{i,i+1}^{incr}$ do
5:         $N_R^{J_i,cur} \leftarrow N_R^{J_i,cur} - 1$
6:     end while
7:     $N_R^{J_i,incr} \leftarrow N_R^{J_i,cur}$
8: end for
```

Algorithm 7 Global optimization strategy for deriving workflow reduce tasks' settings

```
1: $s^{cur} = s^{min} = \{N_R^{J_1,min}, ..., N_R^{J_n,min}\}$
2: $T_{w\_incr} = T_W(s^{min}) \times (1 + w\_increase)$
3: for $i \leftarrow 1$ to $n$ do
4:     $N_R^{J_i,incr} \leftarrow N_R^{J_i,min}$
5: end for
6: while true do
7:     $bestJob = -1, \ \ maxGain = 0$
8:     for $i \leftarrow 1$ to $n$ do
9:         $N_R^{J_i,tmp} \leftarrow N_R^{J_i,incr} - 1$
10:        $s^{tmp} = s^{cur} \cup \{N_R^{J_i,tmp}\} - \{N_R^{J_i,incr}\}$
11:        if $T_W(s^{tmp}) \leq T_{w\_incr}$ then
12:            $Gain = \frac{R_W(s^{min}) - R_W(s^{tmp})}{T_W(s^{tmp}) - T_W(s^{min})}$
13:            if $Gain > MaxGain$ then
14:                $maxGain \leftarrow Gain, \ \ bestJob \leftarrow i$
15:            end if
16:        end if
17:    end for
18:    if $bestJob = -1$ then
19:        break
20:    else
21:        $N_R^{bestJob,incr} \leftarrow N_R^{bestJob,incr} - 1$
22:    end if
23: end while
```

rithm 7. First, we apply the user-specified *w_increase* to determine the targeted completion time T_{w_incr} (line 2). The initial number of reduce task for each job J_i is set to $N_R^{J_i,min}$ (lines 3-5), and then we go through the iteration that at each round estimates the *gain* we can get by decreasing the number of reduce tasks by one for each job J_i. The gain is defined as the total resource savings (the difference of the resource usage before and after decreasing the reduce task number of J_i) divided by the corresponding completion time degrade (the difference of the completion time before and after decreasing the reduce task number of J_i). We aim to identify the job that achieves the highest gain with the decreased amount of reduce tasks while satisfying the targeted overall completion time (lines 8-17). We pick the job which brings the largest gain and decrease its reduce task setting by 1 (line 21). Then the iteration repeats until the number of reduce tasks in any job cannot be further decreased because it would cause a violation of the targeted overall completion time T_{w_incr} (line 11).

We demonstrate the process of our global optimization strategy with a simplified example as follows. Give a MapReduce applicaiton with 3 sequential jobs J_1, J_2 and J_3, Table 5.2 shows for each job pair, the corresponding results when we decrease the reduce task number by 1 for J_1, J_2 and J_3 respectively while the last column shows the gain of each operation. As we can see, job pair (j_2, j_3) has the highest gain and we will decrease the reduce task number for j_2 by 1 in this case. After that, the same process continues until the further decrease of reduce task number for any job will violate the completion time threshold for the entire workflow.

Table 5.2: Example of the global optimization strategy

Job pair	CT_increase	R_saving	Gain
$(J_1; J_2)$	2	1	0.5
$(J_2; J_3)$	3	2	**0.67**
(J_3)	5	2	0.4

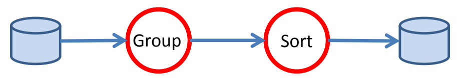
5.3 Evaluation results

We present a set of validation results for the benefits by tuning the jobs settings and experiments that justify the choice of two optimization strategies. The testbed we used is the same as we described in Chapter 3.1.4, but we use a different set of workloads.

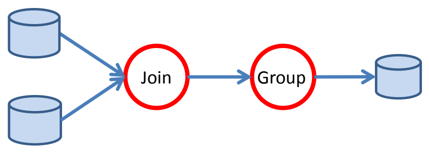
5.3.1 Experimental workloads

To validate the accuracy, effectiveness, and performance benefits of the proposed framework, we use a workload set that consists of queries from TPC-H benchmark and custom queries mining on HP Lab's web proxy log. We provide descriptions of these queries below.

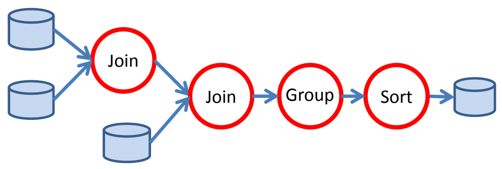
The TPC-H and proxy queries are translated into MapReduce application using *Pig* and are compiled into *sequential* MapReduce workflows that are graphically represented in Figure 5.6.



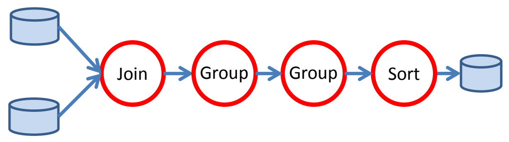
(a) TPC-Q1



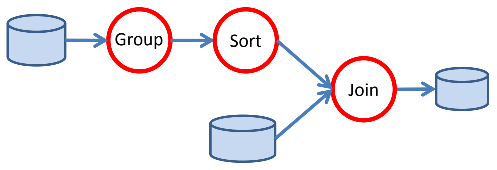
(b) TPC-Q19



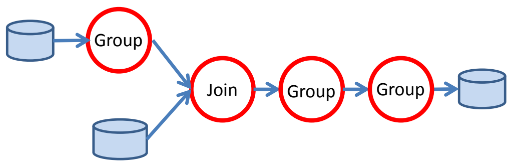
(c) TPC-Q3



(d) TPC-Q13



(e) proxy-Q1



(f) proxy-Q2

Figure 5.6: MapReduce workflows for TPC-H and Proxy queries.

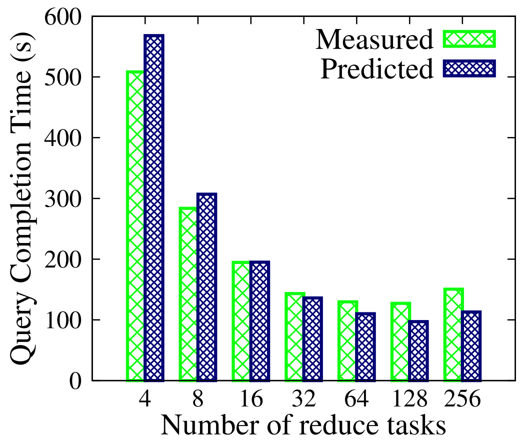
- *TPC-H Q1*: This query provides a summary report of all the *lineitems* shipped as of a given date. The *lineitems* are grouped by different attributes and listed in ascending order. The query is translated into a workflow with two sequential MapReduce jobs as shown in Figure 5.6 (a).

- *TPC-H Q19*: This query reports gross discounted revenue for all orders for three different types of parts that were shipped by air or delivered in person. The query is translated into a workflow with two sequential MapReduce jobs as shown in Figure 5.6 (b).
- *TPC-Q3*: This query retrieves the shipping priority and potential revenue of the orders that had not been shipped and list them in decreasing order of revenue. The query is translated into four jobs as shown in Figure 5.6 (c).
- *TPC-Q13*: This query determines the distribution of customers by the number of orders they have made. (It counts and reports how many customers have no orders, how many have 1, 2, 3, etc.) The query is translated into four jobs as shown in Figure 5.6 (d).
- *proxy-Q1*: This program compares the average daily access frequency for each website during years 2011 and 2012 respectively. The program is translated into three sequential jobs as shown in Figure 5.6 (e).
- *proxy-Q2*: This program computes the intersection between the top 500 popular websites accessed by HP users and the top 100 popular web-sites in US. The program is translated into four jobs as shown in Figure 5.6 (f).

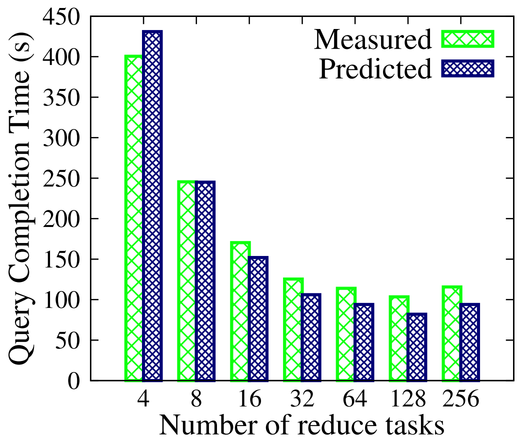
5.3.2 Performance optimization benefits

Since it is infeasible to validate optimal settings by testbed executions (unless we exhaustively execute the programs with all possible settings), we evaluate the models' accuracy to justify the optimal settings procedure and demonstrate the potential benefits with our performance optimizations. In this set of experiments, we use queries *TPC-H Q1* and *TPC-H Q19* from the TPC-H benchmark as our s-tudy cases.

We execute these two queries with the total input size of 10 GB (a scaling factor of 10 using TPC-H data generator) in our 66-node Hadoop cluster. Figure 5.7 shows measured and predicted query completion times for a varied number of reduce tasks in the first job of both workflows (the number of reduce tasks for the second job is fixed in these experiments). First of all, results presented in Figure 5.7 reflect a good quality of our models: the difference between measured and predicted completion times for most of the experiments is less than 10%. Moreover, the predicted completion times accurately reflect a similar trend observed in measured completion times of the studied workflows as a function of the reduce task configuration. These experiments demonstrate that there is a significant difference (up to 4-5 times) in the total completion times depending on the reduce task settings.



(a) *TPC-H Q1*



(b) *TPC-H Q19*

Figure 5.7: Workflow model validation for *TPC-H Q1* and *TPC-H Q19*.

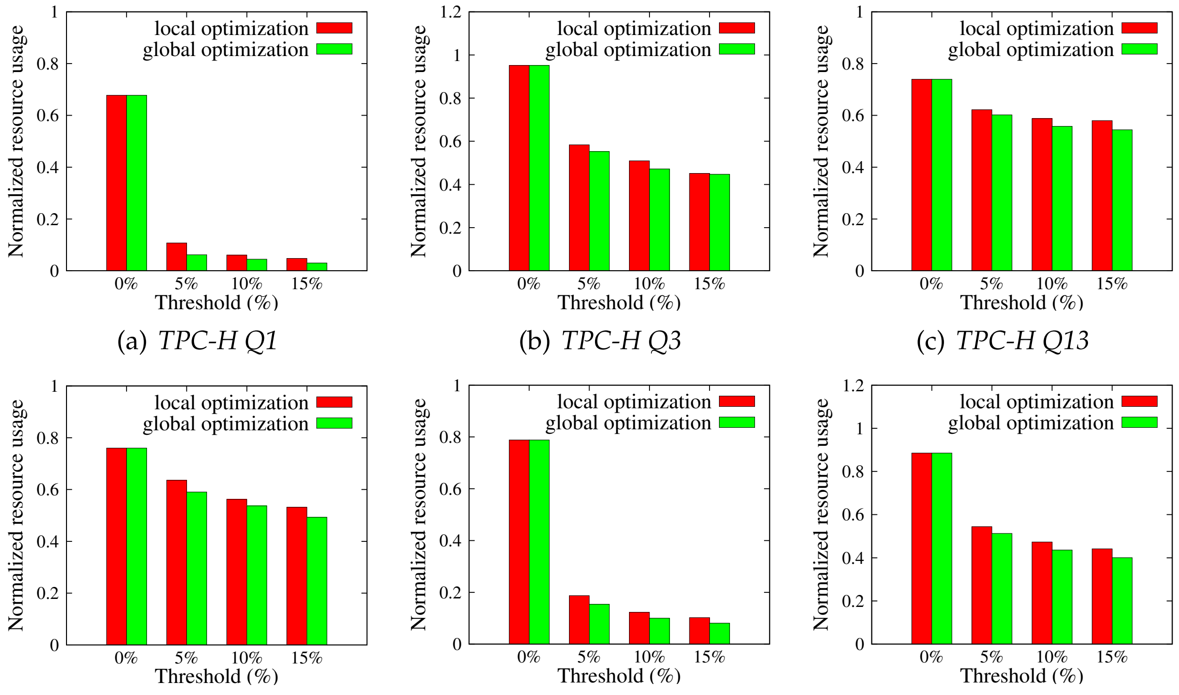
Figure 5.7 shows that the query completion time decreases with the increased number of reduce tasks (because it leads to a higher concurrency degree and a smaller amount of data processed by each task). However, at some point job settings with a high number of reduce tasks (e.g., 256) may have a negative effect due to higher overheads and higher resource allocation required to process such a job.

Another interesting observation from the results in Figure 5.7 is that under two settings with a number of reduce tasks equal to 64 and 128 the total completion times are very similar while the number of required reduce slots for a job execution increases twice. The proposed framework enables the user to identify useful trade-offs in achieving the optimized total completion time while minimizing the amount of resources required for a workflow execution.

5.3.3 Performance benefits of the optimization strategies

In Chapter 5.3.2 and 5.1.1, we show that tuning the reduce task settings in workflows lead to significant performance benefits (2-5 times completion time improvements). Moreover, we observe that a similar overall completion time may be obtained within a certain range of reduce task settings. Figures 5.7-5.3 show that users may achieve nearly optimal completion time (5%-10% of the minimal one) by using a significantly smaller number of reduce tasks (half or less). This can lead to significant resource savings of both map and reduce slots over time. Therefore, tuning the reduce task settings that minimize the overall completion time while optimizing the resources used by the application is a desirable feature for an efficient workload management in the cluster.

We evaluate two optimization strategies introduced in Chapter 5.2.1 for deriving job settings along the workflow and analyzing the achievable performance trade-offs.

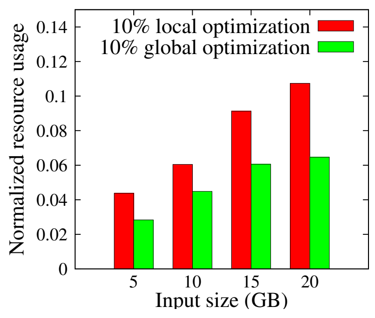


(a) *TPC-H Q1* (b) *TPC-H Q3* (c) *TPC-H Q13*

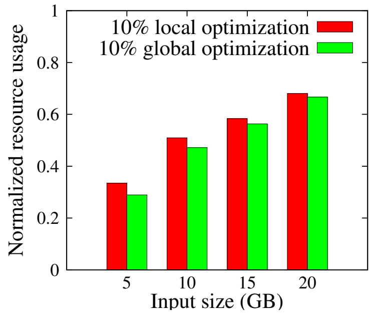
(d) *TPC-H Q19* (e) *Proxy Q1* (f) *Proxy Q2*

Figure 5.8: Local and global optimization strategies: resource usage with different *w_increase* thresholds.

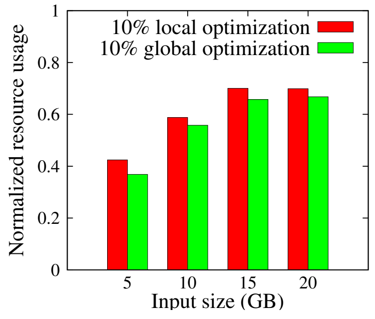
Figure 5.8 presents the normalized resource usage under local and global optimization strategies when they are applied with different thresholds for a overall completion time increase, i.e., *w_increase= 0%, 5%, 10%, 15%*. Figures 5.8 (a)-(d) show the *measured results* for four TPC-H queries with the input size of 10GB (i.e., scaling fastor of 10), and Figures 5.8 (e)-(f) show results for two proxy queries that process 3-month data of web proxy logs. For presentation purposes, we show the normalized resource usage with respect to the resource usage under the *rule of thumb* setting that sets the number of reduce tasks in the job to 90% of the available reduce slots in the cluster. In the presented results, we also eliminate the resource usage of the map stage in the first job of the workflow as its execution does not depend on job settings.



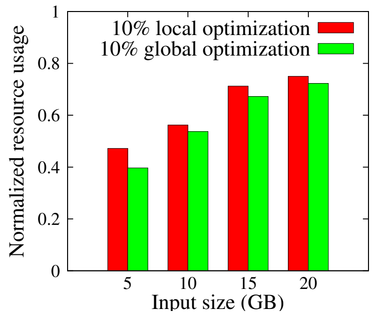
(a) *TPC-H Q1*



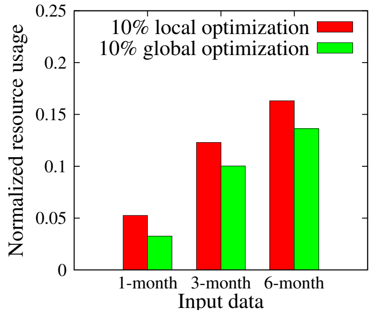
(b) *TPC-H Q3*



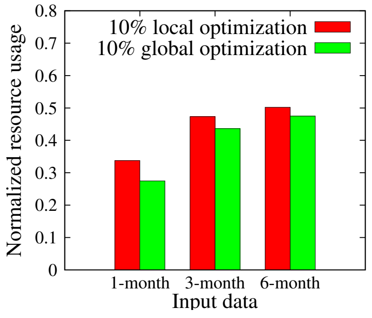
(c) *TPC-H Q13*



(d) *TPC-H Q19*



(e) *Proxy Q1*



(f) *Proxy Q2*

Figure 5.9: Local and global optimization strategies: resource usage with *w_increase=10%* while processing different size input datasets.

The results are quite interesting. The first group of bars in Figure 5.8 shows the normalized resource usage when a user aims to achieve the minimal overall completion time (*w_increase= 0%*). Even in this case, there are 5%-30% resource savings compared to the *rule of thumb* settings. When *w_increase= 0%* the local and global optimization strategies are identical and produce the same results. However, if a user accepts 5% of the completion time increase it leads to very significant resource savings: 40%-95% across different queries shown in Figure 5.8. The biggest resource savings are achieved for *TPC-H Q1* and *Proxy Q1*: 95% and 85% respectively. Moreover, for these two queries global optimization strategy outperforms the local one by 20%-40%. As we can see the performance trade-offs are application dependent.

Figure 5.9 compares the normalized resource usage under local and global optimization strategies when the queries process different amounts of input data (for proxy queries, x-month means that x-months of logs data are used for processing). In these experiments, we set *w_increase=10%*. The results again show that the reduce task settings and related performance benefits are not only application dependent, but also depend on the amount of data processed by the application. The global optimization policy always outperforms the local one, and in some cases, the gain is significant: up to 40% additional resource savings for *TPC-H Q1* and *Proxy Q1* for processing smaller datasets.

In summary, our performance optimization through tuning the job settings offers an automated way for a proactive analysis of achievable performance trade-offs to enable an efficient workload management in a Hadoop cluster.

5.4 Conclusion

Many companies are on a fast track of designing advanced data analytics over large datasets using MapReduce environments. Optimizing the execution efficiency of these applications is a challenging problem that requires the user experience and expertize.

In this chapter, we propose an automatic performance optimization framework for guiding the user efforts of tuning the job settings (i.e., the number of reduce tasks) within MapReduce applications defined as sequential workflows while achieving performance objectives. The proposed approach is based on our performance evaluation framework described in Chapter 3. We design and analyze two optimization strategies for determining the reduce task numbers in a MapReduce workflow that optimize the overall completion time while minimizing the resource usage for executing this workflow The approach does not require any change of the Hadoop or Pig systems and our evaluation results show that in many cases, by allowing 10% increase in the overall completion time, one can gain 40%-90% of resource usage savings.

Chapter 6

Related Work

Performance modeling, resource management, and performance optimization are new topics in MapReduce environments but they have already received much attention. We provide here a summary about the works from each part in the following sections respectively.

6.1 Performance model for MapReduce applications

In the past few years, performance modeling has received much attention and different approaches were offered for predicting performance of MapReduce applications.

ParaTimer [44] and its earlier work *Parallax* [45] offers a progress estimator for estimating the progress of parallel queries expressed as Pig programs that can translate into directed acyclic graphs (DAGs) of MapReduce jobs. Instead of detailed platform performance profiling that is designed in our work, the authors rely on earlier debug runs of the query for estimating throughput of map and reduce stages on the user input data samples. The approach relies on a simplified assumption that map (reduce) tasks of the same job have the same duration. The usage of the FIFO scheduler limits the approach applicability for progress estimation of multiple jobs running in the cluster with a different Hadoop scheduler.

In *Starfish* [28], the authors apply dynamic Java instrumentation to collect a runtime monitoring information about job execution. They create a fine granularity job profile that consists of a diverse variety of metrics. This detailed job profiling enables the authors to predict the job execution under different Hadoop configuration parameters, automatically derive an optimized cluster configuration, and solve cluster sizing problem [27]. However, collecting a large set of metrics comes at a cost, and to avoid a significant overhead, profiling is applied to a small fraction of tasks. Another main challenge outlined by the authors is a design of an efficient searching strategy through the high-dimensional space of parameter values. Our phase profiling approach is inspired by *Starfish* [28]. However, we build a lightweight profiling tool that only collects phase durations and therefore, it can profile each task at a minimal cost. Moreover, the counter-based platform profiling can be done in a small deployment cluster, and it does not impact the production jobs.

Tian and Chen [64] propose predicting a given MapReduce application performance from a set of test runs on small input datasets and a small Hadoop cluster. By executing a variety of 25-60 test runs the authors create a training set for building a model of a given application. Once derived, this model is able to predict the future performance of the same application when executed on a larger input and a larger Hadoop cluster. The limitation for this model is that the model it closely tired with the application characteristic, when given a new application, the model has to be rebuilt using another training set created with the new application.

ARIA [66] builds an automated framework for extracting compact job profiles from the past application run(s). These job profiles form the basis of a *MapReduce analytic performance model* that computes the lower and upper bounds on the job completion time. It also provides an SLO-based scheduler for MapReduce jobs with timing requirements. However, the proposed approach works only for single MapReduce jobs. In [58], the authors design a model based on closed Queuing Networks for predicting the execution time of the map phase of a MapReduce job. The proposed model captures contention at compute nodes and parallelism gains due to the increased number of slots available to map tasks.

Ganapathi et al. [25] use Kernel Canonical Correlation Analysis to predict the performance of MapReduce workloads. However, they concentrate on Hive queries and do not attempt to model the actual execution of the MapReduce job, but discover the feature vectors through statistical correlation.

The problem of predicting the application performance on a new or different hardware has fascinated researchers and been an open challenge for a long time [42, 56]. In 1995, Larry McVoy and Carl Staelin introduced the *lmbench* [42] – a suite of operating system microbenchmarks that provides an extensible set of portable programs for system profiling and the use in cross-platform comparisons. Each microbenchmark was purposely created to capture some unique performance properties and features that were present in popular and important applications of that time. Although such microbenchmarks can be useful in understanding the end-to-end behavior of a system, the results of these microbenchmarks provide little information to indicate how well a particular application will perform on a particular system.

A different approach is to use a set of specially generated microbenchmarks to characterize the *relative performance* of the processing pipelines of two underlying Hadoop clusters: *old* and *new* ones. Herodotou et. al. [27] attempt to derive a *relative model* for Hadoop clusters comprised of different Amazon EC2 instances. They use the Starfish profiling technique and a small set of six benchmarks to exercise job processing with data compression and combiner turned on and off. The model is generated with the M5 Tree Model approach [51].

The other prior examples of successfully building relative models include a relative fitness model for storage devices [43] using CART models, and a relative model between the native and virtualized systems [74] based on a linear-regression technique. The main challenges outlined in [43, 74] for building the accurate models are the tailored benchmark design and the benchmark coverage. Both of these challenges are non-trivial: if a benchmark collection used for system profiling is not representative or complete to reflect important workload properties then the created model might be inaccurate. Finding the right approach to resolve these issues is a non-trivial research challenge.

Besides using performance model in predicting the completion time, a few MapReduce simulators were introduced for the analysis and exploration of Hadoop cluster configuration and optimized job scheduling decisions. The designers of *M-RPerf* [70] aim to provide a fine-grained simulation of MapReduce setups. To accurately model inter- and intra rack task communications over network *MRPerf* uses the well-known ns-2 network simulator. The authors are interested in modeling different cluster topologies and in their impact on the MapReduce job performance. In our work, we follow the directions of *SimMR* simulator [67] and focus on simulating the job master decisions and the task/slot allocations across multiple jobs. We do not simulate details of the TaskTrackers (their hard disks or network packet transfers) as done by *MRPerf*. In spite of this, our approach accurately reflects the job processing because of our profiling technique to represent job latencies during different phases of MapReduce processing in the cluster. *SimMR* is very fast compared to *MRPerf* which deals with network-packet level simulations.

Mumak [10] is an open source Apache's MapReduce simulator. It replays traces collected with a log processing tool, called Rumen [6]. The main difference between Mumak and *SimMR* is that Mumak omits modeling the shuffle/sort phase that could significantly affect the accuracy.

6.2 Resource management for MapReduce jobs

With a primary goal of minimizing the completion times of large batch jobs the simple *FIFO* scheduler (initially used in Hadoop) was quite efficient. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [2] was introduced to support more efficient and flexible cluster sharing. Capacity scheduler partitions the cluster resources into different resource pools and provides separate job queues and priorities for each pool. However, within the pools, there are no additional capabilities for performance management of the jobs.

As a new trend, in current MapReduce deployments, there is an increasing fraction of ad-hoc queries which expect to get quick results back. When these queries are submitted along with long production jobs, neither FIFO or Capacity scheduler works well in these situation. This situation has motivated the design of the *Hadoop Fair Scheduler* (HFS) [77]. It allocates equal shares to each of the users running the MapReduce jobs, and also tries to maximize data locality by delaying the scheduling of the task, if no local data is available. Similar fairness and data locality goals are pursued in *Quincy* scheduler [33] proposed for the Dryad environment [32]. The authors design a novel technique that maps the fair-scheduling problem to the classic problem of min-cost flow in a directed graph to generate a schedule. The edge weights and capacities in the graph encode the job competing demands of data locality and resource allocation fairness. While both HFS and Quincy allow fair sharing of the cluster among multiple users and their applications, these schedulers do not provide any special support for achieving the application performance goals and the service level objectives (SLOs).

A step in this direction is proposed in FLEX [73] which extends HFS by proposing a special slot allocation schema to optimize explicitly some given scheduling metric. FLEX relies on the speedup function of the job (for map and reduce stages) that produces the job execution time as a function of the allocated slots. This function aims to represent the application model, but it is not clear how to derive this function for different applications and for different sizes of input datasets. FLEX does not provide a technique for job profiling and detailed MapReduce performance model, but instead uses a set of simplifying assumptions about the job execution, tasks durations and job progress over time. The authors do not offer a case study to evaluate the accuracy of the proposed approach and models in achieving the targeted job deadlines.

Another interesting extension of the existing Hadoop FIFO and fair-share schedulers using the dynamic proportional sharing mechanism is proposed by Dynamic Priority (DP) scheduler [55]. It allows users to purchase and bid for capacity (map and reduce slots) dynamically by adjusting their spending over time. The authors envision the scheduler to be used by deadline or cost optimizing agents on users behalf. While this approach allows dynamically controlled resource allocation, it is driven by economic mechanisms rather than a performance model and/or application profiling for achieving job completion deadlines.

LATE [79] improves the MapReduce job completion time when executing on heterogeneous environment. It identifies the ”stragglers” (i,e, tasks that make slow progress) and schedule speculative execution for those tasks earlier to eliminate its negative effect on completion time. [38] proposes a general scheduling algorithm which tries to optimize both the completion time and monetary cost in public cloud through a pre-computed matching of the tasks and the computer nodes. Dynamic proportional share scheduler [17] allows users to bid for map and reduce slots by adjusting their spending over time. Similar to Capacity and Fair scheduler, they are not designed for achieving the application completion time goals.

Jockey [23] focus on the Dryad framework and the SCOPE system on top of it. It tries to provide latency SLOs for data parallel jobs by pre-computing the statistics on the job’s remaining run time and corresponding resource allocation with a simulator and then dynamically control the resource allocation during runtime to achieve the jobs completion time target. The limitation of Jockey is that it needs to train(simulate) the completion time distributions for each job and when the input size changes, they system needs to train it again as a new job which leads to scalability problem.

In [49], the authors try to adopt the online scheduling strategy from real-time system like Earliest Deadline First (EDF) that assigns higher priority to a job with a tighter (earlier) deadline. However, it does not provide any guarantees for achieving the job performance goals: the scheduler assigns all available resources to the job with the highest priority (i.e., the earliest deadline) and then kills the job if its deadline can not be satisfied.

Another group of related work is based on resource management that considers monetary cost and budget constraints. In [59], the authors provide a heuristic to optimize the number of machines for a bag of jobs while minimizing the overall completion time under a given budget. This work assumes the user does not have any knowledge about the job completion time. It starts with a single machine and gradually adds more nodes to the cluster based on the average job completion time updated every time when a job is finished. In our approach, we use job profiles for optimizing the job schedule and provisioning the cluster.

In [11], the authors design a budget-driven scheduling algorithm for MapReduce applications in the heterogeneous cloud. They consider iterative MapReduce jobs that take multiple stages to complete, each stage contains a set of map or reduce tasks. The optimization goal is to select a machine from a fixed pool of heterogeneous machines for each task to minimize the job completion time or monetary cost. The proposed approach relies on a prior knowledge of the completion time and cost for a task i executed on a machine j in the candidate set. In our work, we aim at minimizing the makespan of the set of jobs and design an ensemble of methods and tools to evaluate the job completion times as well as their makespans as a function of allocated resources.

In [38], Kllapi et al. propose scheduling strategies to optimize performance/cost trade-offs for general data processing workflows in the cloud. Different machines are modelled as containers with different CPU, memory, and network capacities. The computation workflow contains a set of nodes as operators and edges as data flows. The authors provide a greedy and local search algorithms to schedule operators on different containers so that the optimal performance (cost) is achieved without violating budget or deadline constraints. Compared to our profiling approach, they estimate the operator execution time using the CPU container requirements. This approach does not apply for estimating the durations of map/reduce tasks – their performance depends on multiple additional factors, e.g., the amount of RAM allocated to JVM, the I/O performance of the executing node, etc. The authors present only simulation results without validating the simulator accuracy.

6.3 MapReduce performance optimizations

Though MapRecuce and Hadoop have gained increasing popularity due to its simple programming model, flexible data presentation as well as its scalability and fault tolerance, the execution performance remains a concerns for many users as studies [48] shows that Hadoop query times are 1 to 2 orders of magnitude slower than database system performing the same analysis. How to improve the execution efficiency for processing applications of different types on MapReduce framework has received great attention from both the academical and industrial community.

An important part of the optimization works focus on reducing the cost to transfer data through networks during the execution. *MRShare* [46] and *CoScan* [71] offer the automatic sharing frameworks that merge the executions of MapReduce jobs that have common data inputs in such a way that this data is only scanned once, and the entire completion time is significantly reduced. AQUA [75] proposes an automatic query analyzer for MapReduce applications on relational data analysis. It uses a cost based approach to 1) decide the most efficient strategy for join operation and 2) reconstruct the MapReduce DAGs to minimize the possible intermediate data generated during the workflow execution. In [34], the authors detect from the user code the similar logic of data operations like selection and projection, and then apply the traditional database query optimizations (e.g., B+Trees for selection and column-store-style technique for projection) for those operations.

Another part of the optimization works focus on eliminating the impact of data skew in the reduce stage. In [52], the authors propose a load balancer to balance the reducer work in MapReduce workloads. It uses a progressive sampler to estimate the load associated with each reduce-key and use Key Chopping to split keys with heavy load and Key packing to pack keys with light load. SkewTune [40] focus on reduce the negative affect of completion time caused by skewed data as well as hardware heterogeneity. Specifically, it identifies the tasks that process large amount of data during the execution, then dynamically break those tasks into small sub-tasks and redistribute the sub-tasks to different nodes.

Originally, Hadoop was designed for homogeneous environment. There has been recent interest [79] in heterogeneous MapReduce environments. There is a body of work focusing on performance optimization of MapReduce executions in heterogeneous environments. Zaharia et al. [79], focus on eliminating the negative effect of stragglers on job completion time by improving the scheduling strategy with speculative tasks. The Tarazu project [13] provides a communication-aware scheduling of map computation which aims at decreasing the communication overload when faster nodes process map tasks with input data stored on slow nodes. It also proposes a load-balancing approach for reduce computation by assigning different amounts of reduce work according to the node capacity. Xie et al. [76] try improving the MapReduce performance through a heterogeneity-aware data placement strategy: a faster nodes store larger amount of input data. In this way, more tasks can be executed by faster nodes without a data transfer for the map execution. Polo et al. [50] show that some MapReduce applications can be accelerated by using special hardware. The authors design an adaptive Hadoop scheduler that assigns such jobs to the nodes with corresponding hardware.

Much of the recent work also focuses on anomaly detection [39], stragglers [62] and outliers [14] control in MapReduce environments as well as on optimization and tuning cluster parameters and testbed configuration [31, 36].

These are interesting orthogonal optimization directions that pursue different performance objectives. In our work, we focus on optimizing the execution performance via tuning the number of reduce tasks of its jobs, while keeping the Hadoop cluster configuration unchanged. We are not aware of published papers solving this problem.

Chapter 7

Conclusion

This chapter summarizes the dissertation work and provides a few possible directions for future research work.

7.1 Summary

This dissertation centers around performance modeling and resource management for MapReduce applications. It introduces a performance modeling framework for estimating completion time for complex MapReduce applications defined as a DAG of MapReduce jobs when it is executed on a given platform with different resource allocations and different input data set(s).

Based on the performance modeling framework, we further introduce resource allocation strategies as well as our customized deadline-driven scheduler in estimating and controlling the appropriate amount of resource that should be allocated to each application to meet their (soft) deadlines. For the problem of resource provision in public cloud environment, we identify that different applications show different execution efficiency when executed on different platforms and provide guidance to help users determine the optimal Hadoop cluster deployment according to their workloads.

In addition, the proposed performance modeling framework also enables automated tuning of the job settings (i.e., number of reduce tasks) along the applications defined as sequential MapReduce workflows for optimizing both completion time and the resource usage for the workflows.

7.2 Future work

To conclude this dissertation, we discuss some interesting directions for future research.

Dynamic resource management framework. Building on the current performance modeling framework, I hope to extend it towards a more general resource management and optimization framework which dynamically allocates different types of resources according to the characteristics of MapReduce jobs and different service level objectives (e.g., completion time, cost, energy consumption). The resources considered could be defined in details by its computing, communication and storage capacity and provided by different service providers. The framework should also be able to adapt to the change of workloads and system utilization by dynamically adding or removing available resources in an elastic computing environment.

Generalizing the framework for parallel data processing in distributed systems. Our performance modeling framework is built on the MapReduce and Hadoop architecture. However, the methodology we provided should not be restricted to this specific platform. As the big data problem become more important, new frameworks are evolving for parallel data processing in distributed systems. First the Hadoop is evolving to the next generation called YARN [4, 65], which suppots different processing mode and aims to improve the system utilization. There are projects aims to provide better performance for MapReduce styple proceeding by using in memory file systems like Spark [78], Shark [22] and M3R [57] and there are also I/O efficient large scale data processing system like Sailfish [53] and Themis [54]. As a future work, I plan to extend the existing approaches on different data-parallel middleware platforms in distributed systems and explore the possibility to generalize the framework to support different platforms.

Performance modeling in public cloud with virtualization. Today's public cloud platforms make extensive use of virtualization across computing storage, and network resources. An interesting trend that has emerged in recent years is the virtualization of the network layer, first demonstrated by the use of the OpenFlow [41] API as part of the Software Defined Networking (SDN) stack [37], and more recently, the proposed use of Network Function Virtualization (NFV) to virtualized network services. These new innovations aim at making cloud service deployment easier, but also introduce a new set of challenges related to SLA guarantees in a multi-tenant setting. An interesting avenue of work that I plan to explore, is to develop novel performance models and resource allocation strategies that can take into considerations the high degrees of variance in highly virtualized environments.

Bibliography

[1] Apach Hadoop: TeraGen Class. http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/examples/terasort/TeraGen.html.

[2] Apache Capacity Scheduler Guide. http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html.

[3] Apache hadoop: Best practices and anti-patterns. http://developer.yahoo.com/blogs/hadoop/posts/2010/08/apache_hadoop_best_practices_a/,2010.

[4] Apache Hadoop NextGen MapReduce (YARN). http://hadoop.apache.org/docs/current2/hadoop-yarn/hadoop-yarn-site/YARN.html.

[5] Apache PigMix Benchmark. http://wiki.apache.org/pig/PigMix.

[6] Apache Rumen: a tool to extract job characterization data from job tracker logs. https://issues.apache.org/jira/browse/MAPREDUCE-728.

[7] BTrace: A Dynamic Instrumentation Tool for Java. http://kenai.com/projects/btrace.

[8] Fair Scheduler Guide, http://hadoop.apache.org/docs/r0.20.2/fair_scheduler.html.

[9] Hadoop: Open source implementation of MapReduce. http://hadoop.apache.org/.

[10] Mumak: Map-Reduce Simulator `https://issues.apache.org/jira/browse/MAPREDUCE-728`.

[11] Yang Wang and Wei Shi. On Optimal Budget-Driven Scheduling Algorithms for MapReduce Jobs in the Hetereogeneous Cloud. Technical Report TR-13-02, Carleton University, 2013.

[12] TPC Benchmark H (Decision Support), Version 2.8.0, Transaction Processing Performance Council (TPC), http://www.tpc.org/tpch/, 2008.

[13] Faraz Ahmad, Srimat T. Chakradhar, Anand Raghunathan, and T. N. Vijaykumar. Tarazu: Optimizing MapReduce on heterogeneous clusters. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 61–74, 2012.

[14] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Edward 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, 2010.

[15] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, and Amitanand Aiyer. Apache hadoop goes realtime at facebook. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1071–1080, 2011.

[16] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1):285–296, 2010.

[17] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. of the VLDB Endowment*, 1(2), 2008.

[18] Songting Chen. Cheetah: A high performance, custom data warehouse on top of mapreduce. *Proc. VLDB Endow.*, 3(2):1459–1468, 2010.

[19] Sudipto Das, Yannis Sismanis, Kevin S. Beyer, Rainer Gemulla, Peter J. Haas, and John McPherson. Ricardo: integrating r and hadoop. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 987–998, 2010.

[20] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, 2004.

[21] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[22] Cliff Engle, Antonio Lupher, Reynold Xin, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. Shark: Fast data analysis using coarse-grained distributed memory. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 689–692, 2012.

[23] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, pages 99–112, 2012.

[24] Francesco Fusco, Marc Ph. Stoecklin, and Michail Vlachos. Net-fli: On-the-fly compression, archiving and indexing of streaming network traffic. *Proc. VLDB Endow.*, 3(1-2):1382–1393, 2010.

[25] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. Statistics-driven workload modeling for the cloud. In *Proc. of 5th International Workshop on Self Managing Database Systems (SMDB)*, pages 87–92, 2010.

[26] Alan F. Gates, Olga Natkovich, Shubham Chopra, Pradeep Kamath, Shravan M. Narayanamurthy, Christopher Olston, Benjamin Reed, Santhosh S-

rinivasan, and Utkarsh Srivastava. Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. of the VLDB Endowment*, 2(2):1414–1425, 2009.

[27] Herodotos Herodotou, Fei Dong, and Shivnath Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. of ACM Symposium on Cloud Computing*, SOCC '11, pages 18:1–18:14, 2011.

[28] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of 5th Conf. on Innovative Data Systems Research (CIDR)*, pages 261–272, 2011.

[29] Dustin Hillard, Stefan Schroedl, Eren Manavoglu, Hema Raghavan, and Chirs Leggetter. Improving ad relevance in sponsored search. In *Proceedings of the third ACM international conference on Web search and data mining*, WSDM '10, pages 361–370, 2010.

[30] Paul W. Holland and Roy E. Welsch. Robust regression using iteratively reweighted least-squares. *Communications in Statistics-Theory and Methods*, 6(9):813–827, 1977.

[31] Intel. Optimizing Hadoop Deployment. `http://communities.intel.com/docs/DOC-4218`, 2010.

[32] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis 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, 2007.

[33] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew 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, 2009.

[34] Eaman Jahani, Michael J. Cafarella, and Christopher Ré. Automatic optimization for mapreduce programs. *Proc. VLDB Endow.*, 4(6):385–396, 2011.

[35] S.M. Johnson. Optimal Two- and Three-Stage Production Schedules with Setup Times Included. *Naval Res. Log. Quart.*, 1954.

[36] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. In *Proceedings of the 2009 Conference on Hot Topics in Cloud Computing*, HotCloud'09, 2009.

[37] Hyojoon Kim and Feamster Nick. Improving network management with software defined networking. *Communications Magazine*, 51(2):114–119, 2013.

[38] Herald Kllapi, Eva Sitaridi, Manolis M. Tsangaris, and Yannis Ioannidis. Schedule optimization for data processing flows on the cloud. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, SIGMOD '11, pages 289–300, 2011.

[39] Andy Konwinski, Matei Zaharia, Randy Katz, and Ion Stoica. X-tracing Hadoop. *Hadoop Summit*, 2008.

[40] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome 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, 2012.

[41] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, 2008.

[42] Larry McVoy and Silicon Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the 1996 Annual Conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, 1996.

[43] Michael P. Mesnier, Matthew Wachs, Raja R. Sambasivan, Alice X. Zheng, and Gregory R. Ganger. Modeling the relative fitness of storage. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 37–48, 2007.

[44] Kristi Morton, Magdalena Balazinska, and Dan Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 507–518, 2010.

[45] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of MapReduce pipelines. In *Proceedings of IEEE 26th International Conference on Data Engineering*, ICDE'10, pages 681–684, 2010.

[46] Tomasz Nykiel, Michalis Potamias, Chaitanya Mishra, George Kollios, and Nick Koudas. MRShare: sharing across multiple queries in MapReduce. *Proc. VLDB Endow.*, 3(1-2):494–505, September 2010.

[47] Owen OMalley and Arun C. Murthy. Winning a 60 Second Dash with a Yellow Elephant. Proceedings of sort benchmark., 2009.

[48] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 165–178, 2009.

[49] Linh T.X. Phan, Zhuoyao Zhang, Qi Zheng, Boon Thau Loo, and Insup Lee. An empirical analysis of scheduling techniques for real-time cloud-based data processing. In *Proceedings of the 2011 IEEE International Conference on Service-Oriented Computing and Applications*, SOCA '11, 2011.

[50] Jorda Polo, David Carrera, Yolanda Becerra, Vicenc Beltran, Jordi Torres, and Eduard Ayguade. Performance management of accelerated mapreduce work-

loads in heterogeneous clusters. In *Proceedings of 41st International Conference on Parallel Processing*, ICPP'10, pages 653–662, 2010.

[51] J.R. Quinlan. Learning with continuous classes. In *Proc. of Australian joint Conference on Artificial Intelligence*, 1992.

[52] Smriti R. Ramakrishnan, Garret Swart, and Aleksey 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, 2012.

[53] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian 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, 2012.

[54] Alexander Rasmussen, Vinh The Lam, Michael Conley, George Porter, Rishi Kapoor, and Amin Vahdat. Themis: An i/o-efficient mapreduce. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 13:1–13:14, 2012.

[55] Thomas Sandholm and Kevin Lai. Dynamic Proportional Share Scheduling in Hadoop. *LNCS: Proceedings of the 15th Workshop on Job Scheduling Strategies for Parallel Processing*, 6253:110–131, 2010.

[56] Margo Seltzer, David Krinsky, Keith Smith, and Xiaolan Zhang. The case for application-specific benchmarking. In *Proc. of Workshop on Hot Topics in Operating Systems*, HOTOS'99, pages 102–107, 1999.

[57] Avraham Shinnar, David Cunningham, Vijay Saraswat, and Benjamin Herta. M3r: Increased performance for in-memory hadoop jobs. *Proc. VLDB Endow.*, 5(12):1736–1747, 2012.

[58] Bardhan Shouvik and Menasce Daniel A. Queuing Network Models to Predict the Completion Time of the Map Phase of MapReduce Jobs,. In *In Proc. of the Computer Measurement Group International Conference*, 2012.

[59] João Nuno Silva, Luís Veiga, and Paulo Ferreira. Heuristic for Resources Allocation on Utility Computing Infrastructures. In *Proc. of the 6th Intl. Workshop on Middleware for Grid Computing*, MGC '08, pages 9:1–9:6, 2008.

[60] Jian Tan, Xiaoqiao Meng, and Li Zhang. Delay tails in MapReduce scheduling. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, pages 5–16, 2012.

[61] Jian Tan, Xiaoqiao Meng, and Li Zhang. Performance Analysis of Coupling Scheduler for MapReduce/Hadoop. In *Proceedings of the IEEE International Conference on Computer Communications*, INfOCOM'12, pages 2586–2590, 2012.

[62] Jiaqi Tan, Xinghao Pan, Soila Kavulya, Eugene Marinelli, Rajeev Gandhi, and Priya Narasimhan. Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments. In *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium*, NOMS'10, pages 112–119, 2010.

[63] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive - a Warehousing Solution over a Map-Reduce Framework.

[64] Fengguang Tian and Keke Chen. Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 155–162, 2011.

[65] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah,

Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric 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, 2013.

[66] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing*, ICAC'11, pages 235–244, 2011.

[67] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Play It Again, SimMR! In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 253–261, 2011.

[68] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. Resource Provisioning Framework for MapReduce Jobs with Performance Goals. In *Proceedings of the 12th ACM/IFIP/USENIX Middleware Conference*, Middleware'11, pages 165–186, 2011.

[69] Abhishek Verma, Ludmila Cherkasova, and Roy 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 20th IEEE Intl Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS'12, pages 11–18, 2012.

[70] Guanying Wang, Ali R. Butt, Pandey Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *Proceedings of the 17th IEEE Intl Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, MASCOTS'09, pages 1–11, 2009.

[71] Xiaodan Wang, Christopher Olston, Anish Das Sarma, and Randal 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, 2011.

[72] T. White. *Hadoop:The Definitive Guide*. Page 6,Yahoo Press.

[73] Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. In *Proceedings of the 11th ACM/IFIP/USENIX Middleware Conference*, Middleware'10, pages 1–20, 2010.

[74] Timothy Wood, Ludmila Cherkasova, Kivanc Ozonat, and Prashant Shenoy. Profiling and modeling resource usage of virtualized applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, pages 366–387, 2008.

[75] Sai Wu, Feng Li, Sharad Mehrotra, and Beng Chin Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 12:1–12:13, 2011.

[76] Jiong Xie, Shu Yin, Xiaojun Ruan, Zhiyang Ding, Yun Tian, J. Majors, A. Manzanares, and Xiao Qin. Improving mapreduce performance through data placement in heterogeneous hadoop clusters. In *Proceedings of IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum*, IPDPSW'10, pages 1–9, 2010.

[77] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion 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, 2010.

[78] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion 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, 2010.

[79] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, 2008.

[80] Zhuoyao Zhang, L. Cherkasova, and Boon Thau Loo. Getting more for less in optimized mapreduce workflows. In *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, pages 93–100, May 2013.

[81] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Autotune: Optimizing execution concurrency and resource usage in mapreduce workflows. In *Presented as part of the 10th International Conference on Autonomic Computing*, pages 175–181, San Jose, CA, 2013.

[82] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Benchmarking approach for designing a mapreduce performance model. In *Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*, ICPE '13, pages 253–258, 2013.

[83] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Exploiting cloud heterogeneity for optimized cost/performance mapreduce processing. In *Fourth International Workshop on Cloud Data and Platforms*, CloudDP '14, Amsterdam, the Netherlands, 2014.

[84] Zhuoyao Zhang, Ludmila Cherkasova, and Boon Thau Loo. Optimizing cost and performance trade-offs for mapreduce job processing in the cloud. In *IEEE/IFIP Network Opertions and Management Symposium*, NOMS '14, Krakow, Poland, 2014.

[85] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Automated profiling and resource management of pig programs for meeting service level objectives. In *Proceedings of the 9th International Conference on Autonomic Computing*, ICAC '12, pages 53–62, 2012.

[86] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Meeting service level objectives of pig programs. In *Proceedings of the 2Nd International Workshop on Cloud Computing Platforms*, CloudCP '12, pages 8:1–8:6, 2012.

[87] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Optimizing completion time and resource provisioning of pig programs. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (Ccgrid 2012)*, CCGRID '12, pages 811–816, 2012.

[88] Zhuoyao Zhang, Ludmila Cherkasova, Abhishek Verma, and Boon Thau Loo. Performance modeling and optimization of deadline-driven pig programs. *ACM Trans. Auton. Adapt. Syst.*, 8(3):14:1–14:28, September 2013.

[89] Weizhong Zhao, Huifang Ma, and Qing He. Parallel k-means clustering based on mapreduce. In *Proceedings of the 1st International Conference on Cloud Computing*, CloudCom'09, pages 674–679, 2009.