Chapter 1

Towards Optimal Resource Provisioning for Economical and Green MapReduce Computing in the Cloud

Keke Chen, Shumin Guo, James Powers, Fengguang Tian
Data Intensive Analysis and Computing Lab
Ohio Center of Excellence in Knowledge-enabled Computing (Kno.e.sis)
Department of Computer Science and Engineering
Wright State University, Dayton, OH 45435, USA
Email: {keke.chen, guo.18, powers.4, tian.9}@wright.edu

Running MapReduce programs in the cloud introduces the important problem: how to optimize resource provisioning to minimize the financial charge or job finish time for a specific job? An important step towards this ultimate goal is modeling the cost of MapReduce program. In this chapter, we study the whole process of MapReduce processing and build

up a cost function that explicitly models the relationship among the amount of input data, the available system resources (Map and Reduce slots), and the complexity of the Reduce program for the target MapReduce job. The model parameters can be learned from test runs. Based on this cost model, we can solve a number of decision problems, such as the optimal amount of resources that minimize the financial cost with a job finish deadline, minimize the time under certain financial budget, or find the optimal tradeoffs between time and financial cost. With appropriate modeling of energy consumption of the resources, the optimization problems can be extended to address energy-efficient MapReduce computing. Experimental results show that the proposed modeling approach performs well on a number of tested MapReduce programs in both the in-house cluster and Amazon EC2.

1.1 Introduction

With the deployment of web applications, scientific computing, and sensor networks, a large amount of data can be collected from users, applications, and the environment. For example, user clickthrough data has been an important data source for improving web search relevance [9] and for understanding online user behaviors [20]. Such datasets can be easily in terabyte scale; they are also continuously produced. Thus, an urgent task is to efficiently analyze these large datasets so that the important information in the data can be promptly captured and understood. As a flexible and scalable parallel programming and processing model, recently MapReduce [5] (and its open source implementation Hadoop) has been widely used for processing and analyzing such large scale datasets [18, 8, 17, 11, 4, 15].

On the other hand, data analysts in most companies, research institutes, and government agencies have no luxury to access large private Hadoop/MapReduce clouds. Therefore, running Hadoop/MapReduce on top of a public cloud has become a realistic option for most users. In view of this requirement, Amazon has developed Elastic MapReduce[1] that runs on-demand Hadoop/MapReduce clusters on top of Amazon EC2 nodes. There are also scripts[2] for users to manually setup Hadoop/MapReduce on EC2 nodes.

[1] aws.amazon.com/elasticmapreduce/.
[2] wiki.apache.org/hadoop/AmazonEC2

However, running a Hadoop cluster on top of the public cloud has different requirements from running a private Hadoop cluster. First, for each job normally a dedicated Hadoop cluster will be started on a number of virtual nodes to take advantage of the "pay-as-you-use" economical cloud model. Because users' data processing requests are normally coming in intermittently, it is not economical to maintain a constant Hadoop cluster like private Hadoop clusters do. Instead, on-demand clusters are more appropriate to most users. Therefore, there is no multi-user or multi-job resource competition happening within such a Hadoop cluster. Second, it is now the user's responsibility to set the appropriate number of virtual nodes for the Hadoop cluster. The optimal setting may differ from application to application and depend on the amount of input data. An effective method is needed to help the user make this decision.

The problem of optimizing resource provisioning for MapReduce programs involves two intertwined factors: the cost of provisioning the virtual nodes and the time to finish the job. Intuitively, with a larger amount of resources, the job can take shorter time to finish. However, resources are provisioned at cost, which are also related to the amount of time for using the resources. Thus, it is tricky to find the best setting that minimizes the financial cost. With other constraints such as a deadline or a financial budget to finish the job, this problem appears more complicated. More generally, energy consumption of a MapReduce program can also be modeled in a similar way, which is critical to energy efficient cloud computing [2].

We propose a method to help users make the decision of resource provisioning for running MapReduce programs in public clouds. This method is based on the proposed specialized MapReduce cost model that has a number of model parameters to be determined for a specific application. The model parameters can be learned with test runs on a small scale of virtual nodes and small test data. Based on the cost model and the estimated parameters, the user can find the optimal setting for resources by solving certain optimization problems.

Our approach has several unique contributions.

- Different from existing work on the performance analysis of MapReduce program, our approach focuses on the relationship among the critical variables: the number of

Map/Reduce slots, the amount of input data, and the complexity of application-specific components. The resulting cost model can be represented as a weighted linear combination of a set of non-linearly functions of these variables. Linear models provide robust generalization power that allows one to determine the weights with the data collected on small scale tests.

- Based on this cost model, we formulate the important decision problems as several optimization problems. The resource requirement is mapped to the number of Map/Reduce slots; the financial cost of provisioning resources is the product of the cost function and the acquired Map/Reduce slots. With the explicit cost model, the resultant optimization problems are easy to formulate and solve.

- We have conducted a set of experiments on both the local Hadoop cluster and Amazon EC2 to validate the cost model. The experiments show that this cost model fits the data collected from four tested MapReduce programs very well. The experiment on model prediction also shows low error rates.

The entire paper is organized as follows. In Section 1.2, we introduce the MapReduce Programming model and the normal setting for running Hadoop on the public cloud. In Section 1.3, we analyze the execution of MapReduce program and propose the cost model. In Section 1.4, we describe the statistical method to learn the model for a specific MapReduce program. In Section 1.5, we formulate several problems on resource provisioning as optimization problems based on the cost model. In Section 1.6, we present the experimental results that validate the cost model and analyze the modeling errors. In Section 1.7, the related work on MapReduce performance analysis is briefly discussed.

1.2 Preliminary

MapReduce programming for large-scale parallel data processing was recently developed by Google [5] and has become popular for big-data processing. MapReduce is more than a programming model - it also includes the system support for processing MapReduce jobs in

parallel in a large-scale cluster. Apache Hadoop is the most popular open source implementation of the MapReduce framework. Thus, our discussions, in particular the experiments, will be based on Apache Hadoop, although the analysis and modeling approach should also fit other MapReduce implementations.

It is better to understand how MapReduce programming works with an example - the famous WordCount program. WordCount counts the frequency of word in a large document collection. Its Map program partitions the input lines into words and emits tuples $\langle w, 1\rangle$ for aggregation, where 'w' represents a word and '1' means the occurrence of the word. In the Reduce program, the tuples with the same word are grouped together and their occurrences are summed up to get the final result.

Algorithm 1 The WordCount MapReduce program

```
1: map(file)
2: for each line in the file do
3:     for each word w in the line do
4:         Emit(⟨w, 1⟩)
5:     end for
6: end for

1: reduce(w, v)
2: w: word, v: list of counts.
3: d ← 0;
4: for each v_i in v do
5:     d ← d + v_i;
6: end for
7: Emit(⟨w, d⟩);
```

When deploying a Hadoop cluster in a public cloud, users need to request a number of virtual machines from the cloud and then start them with a system image that has the Hadoop package pre-installed. Because users' data may reside in the cloud storage system, e.g., Amazon S3, the Hadoop cluster needs to load the data from the storage system or to be appropriately configured to directly use the storage system. The configuration files are passed to the corresponding master and slave nodes, and the Hadoop cluster can then be started. Here comes the difficult decision problem for the user: how many nodes would be appropriate for a specific job, which will minimize the financial charge and guarantee the job to be finished on time? We start exploring this problem with an analysis on the cost model of MapReduce processing.

1.3 Resource-Time Cost Model for MapReduce

In this section, we analyze the components in the whole MapReduce execution process and derive a cost model in terms of the input data, the application-specific complexity, and the available system resources. The goal of developing this cost model is to identify the relationships (functions) between the resources and time complexity for a specific application. We will see that with this cost model, solving the resource prediction and optimization problems becomes easy.

Due to the complex multi-tenant run-time environment, and uncertain factors such as network traffic and disk I/O performance, it is impossible to precisely model the cost of a MapReduce program. Instead, we will introduce a statistical modeling approach to minimize the possible modeling error. We will give the basic idea of modeling. Let the amount of system resources be S, which can be the number of virtual machines of certain type in a public cloud. Let the amount of input data be D, and the MapReduce setting be C, e.g., the number of Reduce tasks in our discussion. We want to find a cost function - the total time cost of the MapReduce job $T = T(S, D, C)$. There are a number of special features with this modeling task.

1. Because Map/Reduce tasks have very different logic and time complexity for different applications, this cost function should be different from application to application.
2. Ideally, this cost function should be learned from small-scale instances that have small amounts of resources and input data, and still be robust for large-scale instances. It can certainly provide better models for repetitively running jobs.
3. We expect to learn a closed-form function, which can be nicely incorporated in optimization tasks. Some machine learning methods [6] result in special forms of function such as decision trees, which are not easy to handle in optimization and will not serve our purpose well.

Because of these special requirements, we aim to design a modeling method that gives a closed form function with good generalization power. There are two general ways to do cost

modeling. First, we can carefully analyze all the components of the system in detail and then try to precisely model the cost functions. As we have already known, this approach is impractical because of the uncertain environment and the diversity of program logic. The other approach is solely depending on learning algorithms to find the cost function. However, due to the limited features (e.g., only four features as we will show), this approach tends to overfit the cost model [6].

We take a combined approach instead. This method depends on the best-effort analysis of the whole process of MapReduce processing framework, which will result in the following cost function

$$T(S, D, C) = \sum_{i=1}^{k} \beta_i h_i(S, D, C) + \beta_0, \tag{1.1}$$

where $h_i(S, D, C)$ are possibly some non-linear transformations of the input factors S, D, and C, which are the time complexity of sequential processing components in the system, and β_i are the component weights, different from application to application. $h_i(S, D, C)$ are obtained through the analysis of the MapReduce processing components, while β_i will be learned for a specific application based on sample runs of that application. With this modeling idea in mind, in the following subsections, we will conduct the modeling analysis, give a concrete formulation of the cost functions of Map task and Reduce task to find $h_i(S, D, C)$, and finally integrate these components into the whole cost function.

1.3.1 Analyzing the Process of MapReduce

MapReduce processing is a mix of sequential and parallel processing. The Map phase is executed before the Reduce phase[3], as Figure 1.1 shows. However, in each phase many Map or Reduce processes are executed in parallel. To clearly describe the MapReduce execution, we would like to distinguish the concepts of *Map/Reduce slot* and *Map/Reduce process*. Each Map (or Reduce) process is executed in a Map (or Reduce) slot. A slot is a unit of computing resources allocated for the corresponding process. According to the system capacity, a computing node can only accommodate a fixed number of slots so that

[3]The Copy operation in the Reduce phase overlaps the Map phase - when a Map's result is ready, Copy may start immediately.

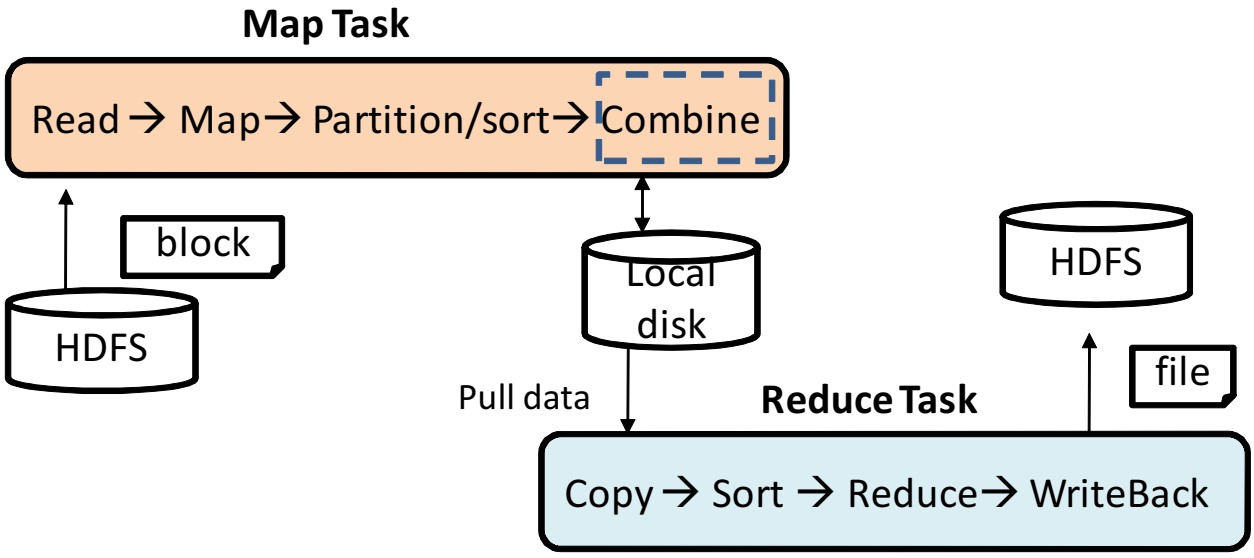

Figure 1.1: Components in Map and Reduce tasks and the sequence of execution.

the parallel processes can be run in the slots without serious competition. In Hadoop, the Tasktracker running in each slave node has to set the number of Map slots and the number of Reduce slots. A common setting for a multi-core computer is to have two Map and Reduce slots per core. Without loss of generality, let's assume there are m Map slots and r Reduce slots in total over all slave nodes.

We define a Map/Reduce process as a Map/Reduce task running on a specific slot. By default, in Hadoop each Map process handles one chunk of data (e.g., 64MB). Therefore, if there are M chunks of data, M Map processes in total will be scheduled and assigned to the m slots. In the ideal case, m Map processes can run in parallel in the m slots - we call it one round of Map processes. If $M > m$, which is normal for large datasets, $\lceil M/m \rceil$ Map rounds are needed.

Different from the total number of Map processes, the number of Reduce processes, denoted as R, can be set by the user or determined by specific application requirements. The Map outputs, i.e., the key-value pairs, are organized by the keys and then distributed evenly by the keys to the R Reduce processes[4]. Similarly, if $R > r$, more than one round of Reduce processes are scheduled. It is probably not very helpful to set a R greater than r because

[4]Thus, it is not meaningful to set R greater than the number of output keys of Map.

there is no restriction on the amount of data a Reduce process can handle. As a rule of thumb, when the number of Map output keys is much large than r, R is often set close to the number of all available Reduce slots for an in-house cluster, e.g., 95% of all Reduce slots [22]. When it comes to public clouds, we will set $R = r$ and choose an appropriate number of Reduce slots, r, to find the best tradeoff between the time and the financial cost.

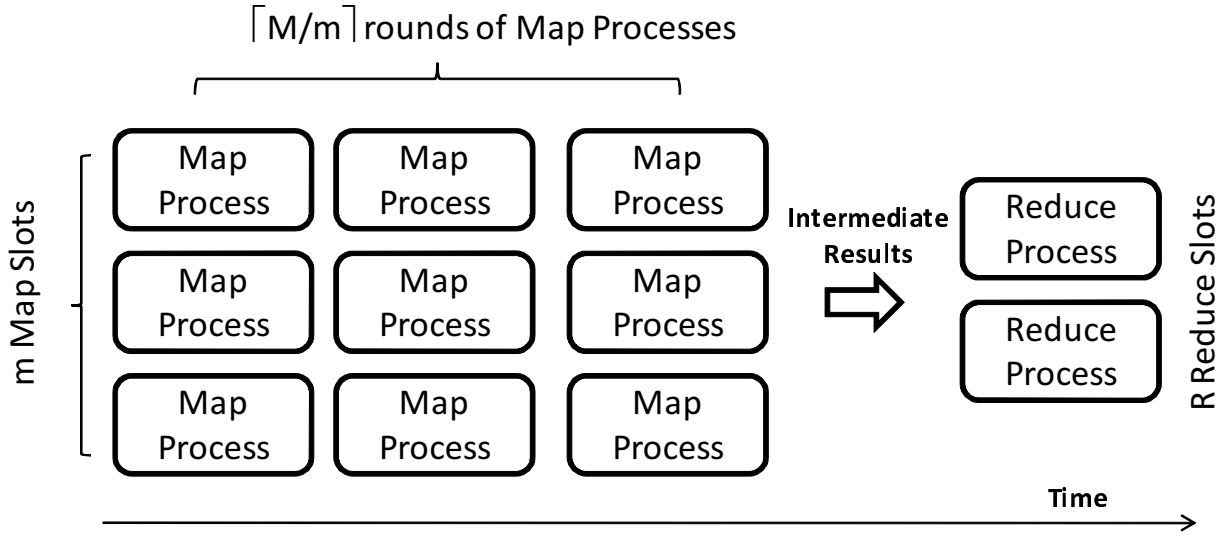

Figure 1.2: Illustration of parallel and sequential execution in the ideal situation.

Figure 1.2 illustrates the scheduling of Map and Reduce processes to the Map and Reduce slots in the ideal situation. In practice, Map processes in the same round may not finish exactly at the same time - some may finish earlier or later than others due to the system configuration, the disk I/O, the network traffic, and the data distribution. But we can use the total number of rounds to roughly estimate the total time spent in the Map phase. The variance caused by all these factors will be considered in modeling. Intuitively, the more available slots, the faster the whole MapReduce job can be finished. However, in the pay-as-you-go setting, there is a tradeoff between the amount of resources and the amount of time to finish the MapReduce job. Thus, we cannot simply increase the amount of resources.

In addition to the cost of Map and Reduce processes, the system has some additional cost for managing and scheduling the M Map processes and the R Reduce processes, which will also be considered in modeling. Based on this understanding, we will first analyze the cost of each Map process and Reduce process, respectively, and then derive the overall cost

model.

1.3.2 Cost of Map Process

A Map process can be divided into a number of sequential components, including Read, Map, Sort/Partition, and optionally Combine, as Figure 1.1 shows. We understand this process in term of a data flow - data sequentially flow through each component and the cost of each component depends on the amount of input data.

The first component is reading a block of data from the disk, which can be either local or remote data block. Let's assume the *average cost* is a function of the size of data block b: $i(b)$.

The second component is the user defined Map program, the time complexity of which is determined by the input data size b, denoted as $f(b)$. The Map program may output data in size of $o_m(b)$ that might vary depending on the specific data. The output will be a list of $\langle key, value \rangle$ pairs.

The result will be partitioned and sorted by the key into R shares for the R Reduce processes. We denote the cost of partitioning and sorting with $s(o_m(b), R)$. If the partitioning process uses a hash function to map the keys, the partitioning cost is independent of R. However, the sorting phase is still affected by R. Let's skip the Combiner component temporarily and we will revisit the Combiner component later.

In summary, the overall cost of a Map process is the sum of the costs (without the Combiner component):

$$\Phi_m = i(b) + f(b) + s(o_m(b), R) + \epsilon_m. \tag{1.2}$$

$i(b)$ and $f(b)$ are only related to the size of the data block b and the complexity of the Map program, independent of the parameters m and M. ϵ_m has a mean zero and some variance σ_m^2, which needs to be calibrated by experiments. We also observed that $s(o_m(b), R)$ is

slightly linear to R. In practice, we can model it with parameters m, M, r, R as

$$\Phi_m(m, M, r, R) = \mu_1 + \mu_2 R + \epsilon_m, \tag{1.3}$$

where μ_1, μ_2, and the distribution of ϵ_m are constants and specific to each application.

1.3.3 Cost of Reduce Process

The Reduce process has the components: Copy, MergeSort, Reduce and WriteResult. These components are also sequentially executed in the Reduce process.

Assume that the k keys of the Map result are equally distributed to the R Reduce processes[5]. In the Copy component, each Reduce process pulls its shares, i.e., k/R keys and the corresponding records, from the M Map processes' outputs. Thus, the total amount of data in each Reduce will be

$$b_R = M \cdot o_m(b) \cdot k/R. \tag{1.4}$$

Here, we simplify the analysis by assuming the amount of data is proportional to the number of keys assigned to the reduce. In practice, many applications have skewed data distributions, i.e., some keys may have more records while other may have less, which may affect the quality of modeling.

The Copy cost is linear to b_R, denoted as $c(b_R)$. However, most of the time is overlapped with the Map phase. Normally only the last few rounds of Map processing may contribute the overall time cost. We thus approximate the cost as $c(b_R) \sim m \cdot o_m(b) \cdot k/R$.

A Merge process follows to merge the M shares from the Map results. Because the records are already sorted by the key, this process simply merges the shares by the key in multiple rounds. Assume the buffer size is B, the Merge round i will generate M/B^i files, and its cost is proportional to b_R. The total number of rounds is $\lceil \log_B M \ rceil$. Thus, the total Merge cost $ms(b_R)$ is proportional to $b_R \lceil \log_B M \rceil$.

[5]For this reason, the user normally selects R to satisfy $k \geq R$. If $R > k$, only k Reduces are actually used.

The Reduce program will process the data with some complexity $g(b_R)$ that depends on the specific application. Assume the output data of the Reduce program has an amount $o_r(b_R)$, which is often less than b_R. Finally, the result is duplicated and written back to multiple nodes, with the complexity linear to $o_r(b_R)$, denoted as $wr(o_r(b_R))$.

In summary, the cost of the Reduce process is the sum of the component costs,

$$\Phi_r = c(b_R) + ms(b_R) + g(b_R) + wr(o_r(b_R)) + \epsilon_r, \tag{1.5}$$

Both the Copy and the WriteResult costs may vary because of the varying network I/O performance, which are modeled with the random variable ϵ_r. Similar to ϵ_m for the Map phase, ϵ_r has a mean zero and some variance σ_r^2. These variances should be captured in modeling.

If we model Φ_r with m, M, r, R, and keep the relevant components for each phase, we have

$$\Phi_r(m, M, r, R) = \lambda_1(m/R) + \lambda_2(M \log M/R) + g(M/R) + \lambda_3(M/R) + \epsilon_r, \tag{1.6}$$

where λ_i and the distribution of ϵ_r are application-specific constants.

1.3.4 Putting All Together

According to the parallel execution model we described in Figure 1.2, the overall time complexity T depends on the number of Map rounds and Reduce rounds. The cost of managing and scheduling the Map and Reduce processes $\Theta(M, R) = \xi_1 M + \xi_2 R$ is linear to M and R, as stated in the documentation [22]. By assuming all the processes in each Map (or Reduce) round finish around the same time, we can represent the overall cost as

$$T = \lceil \frac{M}{m} \rceil \Phi_m + \lceil \frac{R}{r} \rceil \Phi_r + \Theta(M, R). \tag{1.7}$$

We are more interested in the relationship among the total time T, the input data size $M \times b$, the user defined number of Reduce processes R, and the number of Map and Reduce slots, m and r.

This general representation can be slightly simplified with a number of settings. As we have discussed, it is safe to assume $R = r$, as running Reduces in multiple rounds might be unnecessary. Thus, $\lceil \frac{R}{r} \rceil \Phi_r = \Phi_r$. To make it more convenient to manipulate the equation, we also remove $\lceil \rceil$ from $\lceil M/m \rceil$ by assuming $M \geq m$ and M/m is an integer. After plugging in the equations 1.3 and 1.6 and keeping only the variables M, R, and m in the cost model, we get the detailed model

$$\begin{aligned} &T_1(M, m, R) = \\ &\beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{MR}{m} + \beta_3 \frac{m}{R} \\ &+\beta_4 \frac{M \log M}{R} + \beta_5 M/R + \beta_6 M + \beta_7 R + \beta_8 g(\frac{M}{R}) + \epsilon, \end{aligned} \tag{1.8}$$

where β_i are the positive constants specific to each application. Note that $T_1(M, m, R)$ is not linear to its variables, but it is linear to the transformed components: M/m, MR/m, m/R, $M \log M/R$, M/R, M, R, and $g(M/R)$. The parameter β_i defines the contribution of each components in the model. β_0 represents some constant cost invariant to the parameters. β_i are the weights of each components derived in the component-wise cost analysis. Finally, ϵ represents the overall noise. We leave the discussion on the item $g(M/R)$ later.

With Combiner. In the Map process, the Combiner program is used to aggregate the results by the key. If there are k keys in the Map output, the Combiner program reduces the Map result to k records. The cost of Combiner is only subject to the output of the Map program. Thus, it can be incorporated into the parameter β_1. However, the Combiner function reduces the output data of the Map process and thus affects the cost of the Reduce phase. With the Combiner, the amount of data that a Reduce process needs to pull from the Map is changed to

$$b_R = Mk/R. \tag{1.9}$$

Since the item M/R is still there, the cost model (Equation 1.8) applies without any change.

Function g(). The complexity of Reduce program has to be estimated with the specific application. There are some special cases that the $g()$ item can be removed from Equation 1.8. If $g()$ is linear to the size of the input data, then its contribution can be merged to the factor β_4, because $g(M/R) \sim M/R$. For other cases that cannot be merged, a new item should be created and in the cost model. In the linear case, which is common as we have observed, the cost model can be further simplified to

$$\begin{aligned} &T_2(M, m, R) = \\ &\beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{MR}{m} + \beta_3 \frac{m}{R} \\ &+\beta_4 \frac{M \log M}{R} + \beta_5 M/R + \beta_6 M + \beta_7 R + \epsilon, \end{aligned} \tag{1.10}$$

1.4 Learning the Model

With the formulation of the cost function in terms of input variables M, m, and R, we need to learn the parameters β_i. Note that β_i should be different from application to application. We design a learning procedure as follows.

First, for a specific MapReduce program, we randomly choose the variables M, m, and R from certain ranges. For example, m and R (i.e., r) are chosen within 50; M is chosen so that at least two rounds of Map processes are available for testing. Second, we collect the time cost of the test run of the MapReduce job for each setting of (M, m, R), which forms the training dataset. Third, regression modeling [14] is applied to learn the model from the training data with the transformed variables

$$x_1 = M/m, x_2 = MR/m, x_3 = m/R, x_4 = (M \log M)/R, x_5 = M/R, x_6 = M, x_7 = R. \tag{1.11}$$

Because β_i has practical meaning, i.e., the weights of the components in the total cost, we have $\beta_i \geq 0$, $i = 0 \ldots r$, which requires non-negative linear regression [14] to solve the learning problem. The cross-validation method [6] is then used to validate the performance of the learned model. We will show more details in experiments.

1.5 Optimization of Resource Provisioning

With the cost model we are now ready to find the optimal settings for different decision problems. We try to find the best resource allocation for three typical situations: (1) with a certain limited amount of financial budget; (2) with a time constraint; (3) and the optimal tradeoff curve without any constraint. In the following, we formulate these problems as optimization problems based on the cost model.

In all the scenarios we consider, we assume the model parameters β_i have been learned with sample runs in small scale settings. For the simplicity of presentation, we assume the simplified model T_2 (Eq. 1.10) is applied. Cost models with other Reduce complexity do not change the optimization algorithm. Since the input data is fixed for a specific MapReduce job, M is a constant. We also consider all general MapReduce system configurations have been optimized via other methods [1, 8, 7] and fixed for both small and large scale settings. With this setup, the time cost function becomes

$$T_3(m, R) = \alpha_0 + \frac{\alpha_1}{m} + \frac{\alpha_2 R}{m} + \frac{\alpha_3 m}{R} + \frac{\alpha_4}{R} + \alpha_5 R \tag{1.12}$$

where

$$\begin{aligned}
\alpha_0 &= \beta_0 + \beta_6 M, \\
\alpha_1 &= \beta_1 M, \\
\alpha_2 &= \beta_2 M, \\
\alpha_3 &= \beta_3, \\
\alpha_4 &= \beta_4 M \log M + \beta_5 M, \\
\alpha_5 &= \beta_7.
\end{aligned}$$

In the virtual machine (VM) based cloud infrastructure (e.g., Amazon EC2), the cost of cloud resources is calculated based on the number of VM instances used in time units (typically in hours). Let's consider the same type of VM instances are used in the deployment. According to the capacity of a virtual machine (CPU cores, memory, disk and network

bandwidth), a virtual node can only have a fixed number of Map/Reduce slots. Let's denote the number of slots per node as γ, which are also fixed for learning and applying the model. Thus, the total number of slots $m + r$ required by a on-demand Hadoop cluster can be roughly transformed to the number of VMs, v, as

$$v = (m + r)/\gamma. \tag{1.13}$$

If the price of renting one VM instance for an hour is u, the total financial cost is determined by the result $uvT_3(m, R)$. Since we usually set R to r, it follows that the total financial cost for renting the Hadoop cluster is

$$uvT_3(m, R) = u(m + R)T_3(m, R)/\gamma. \tag{1.14}$$

Now we are ready to formulate the optimization problems.

- Given a financial budget ϕ, the problem of finding the best resource allocation to minimize the job time can be formulated as

$$\begin{aligned} &minimize\ T_3(m, R) \\ &subject\ to\ u(m + R)T_3(m, R)/\gamma \leq \phi, \\ &\quad m > 0,\ and\ R > 0. \end{aligned} \tag{1.15}$$

- If the constraint is about the deadline τ for finishing the job, the problem of minimizing the financial cost can be formulated as

$$\begin{aligned} &minimize\ u(m + R)T_3(m, R)/\gamma \\ &subject\ to\ T_3(m, R) \leq \tau, m > 0,\ and\ R > 0. \end{aligned} \tag{1.16}$$

- The above optimization problem can also be slightly changed to describe the problem that the user simply wants to find the most economical solution for the job without the deadline, i.e., the constraint $T_3(m, R) \leq \tau$ is removed.

Note that the T_3 model parameters might be specific for a particular type of VM instance that determines the parameters u and γ. Therefore, by testing different types of VM instance and applying this optimization repeatedly on each instance type, we can also find which instance type is the best.

These optimization problems do not involve complicated parameters except for the T_3 function. Once we learn the concrete setting of the T_3 model parameters, these optimization problems can be nicely solved since they are all in the category of well-studied optimization problems. There are plenty of papers and books discussing how to solve these optimization problems. In particular, the search space of m and R is quite limited, for many medium-scale MapReduce jobs, they are normally integers less than 10,000. In this case, a brute-force search over the entire space to find the optimal result will not cost much time. Therefore, we will skip the details of solving these problems.

1.6 Experiments

As we have shown, as long as the cost model is accurate, the optimization problems are easy to solve. Therefore, our focus of experiments will be validating the formulated cost model. We first describe the setup of the experiments, including the experimental environment and the datasets. Four programs are presented: WordCount, TeraSort, PageRank and Join, which are used in evaluating the cost model. Finally, a restrict evaluation on both the in-house cluster and Amazon Cloud will be conducted to show the model goodness of fit and the prediction accuracy.

1.6.1 Experimental Setup

The experiments are conducted in our in-house 16-node Hadoop cluster and Amazon EC2. We describe the setup of the environments and the datasets used for experiments as follows.

In-house Hardware and Hadoop Configuration. Each node in the in-house cluster

has four quad-core 2.3Mhz AMD Opteron 2376, 16GB memory, and two 500GB hard drives, connected to other nodes with a gigabit switch. Hadoop 1.0.3 is installed in the cluster. One node serves as the master node and 15 nodes as the slave nodes. The single master node runs the *JobTracker* and the *NameNode*, while each slave node runs both the *TaskTracker* and the *DataNode*. Each slave node is configured with eight Map slots and six Reduce slots (about one process per core). Each Map/Reduce process uses 400MB memory. The data block size is set to 64 MB. We use the Hadoop fair scheduler[6] to control the total number of Map/Reduce slots available for different testing jobs.

Amazon EC2 Configuration. We also used the on-demand clusters provisioned from Amazon EC2 for experiments. Only the small instances (1EC2 compute unit, 1.7GB memory, and 160GB hard drive) are used to setup the on-demand clouds. For the simplicity of configuration, one Map slot and one Reduce slot share one instance. Therefore, a cluster that needs m Map slots and r Reduce slots will need $\max\{m, r\} + 1$ small instances in total, with the additional instance as the master node. The existing script[7] in the Hadoop package is used to automatically setup the required Hadoop cluster (with proper node configurations) in EC2.

Datasets. We use a number of generators to generate three types of testing datasets for the testing programs. (1) We revise the RandomWriter tool in the Hadoop package to generate random float numbers. This type of data is used by the Sort program. (2) We also revise the RandomTextWriter tool to generate text data based on a list of 1000 words randomly sampled from the system dictionary /usr/share/dict/words. This type of data is used by the WordCount program and the TableJoin program. (3) The third dataset is a synthetic random graph dataset, which is generated for the PageRank program. Each line of the dataset starts with a node ID and its initial PageRank, followed by a list of node IDs representing the node's outlinks. Both the node ID and the outlinks are randomly generated integers.

Each type of data consists of 150 1GB files. For a specific testing task with the predefined size of input data (the parameter M), we will randomly choose the required number of files

[6]http://hadoop.apache.org/docs/r1.1.1/fair_scheduler.html
[7]wiki.apache.org/hadoop/AmazonEC2

from the pool to simulate input data.

Modeling Tool. As we mentioned, we will need a regression modeling method that works on the constraints $\beta_i \geq 0$. In experiments, we use the matlab function `lsqnonneg`[8] to learn the model, which squarely fits our goal.

1.6.2 Testing Programs.

In this section, we describe the MapReduce programs used in testing and give the complexity of each one's Reduce program, i.e., the $g()$ function. If $g()$ is in one of the two special cases, the simplified cost model Eq. 1.10 is used.

WordCount is a sample MapReduce program in the Hadoop package. The Map program splits the input text into words and the result is locally aggregated by word with a Combiner; the Reduce program sums up the local aggregation results $\langle word, count \rangle$ by words and output the final word counts. Since the number of words is limited, the amount of output data to the Reduce stage and the cost of Reduce stage are small, compared to the data and the processing cost for the Map stage. The complexity of the Reduce program, $g()$, is linear to Reduce's input data.

Sort is also a sample MapReduce program in the Hadoop package. It depends on a custom partitioner that uses a sorted list of $N-1$ sampled keys to define the key range for each Reduce process. All keys such that sample$[i-1]$ <= key < sample$[i]$ are sent to Reduce i. Then, the inherent MergeSort in the Shuffle stage sorts the input data to the Reduce. This guarantees that the output of Reduce i are all less than the output of Reduce i+1. Both the Map program and the Reduce program do nothing but simply pass the input to the output. Therefore, the function $g()$ is also linear to the size of the input of Reduce.

PageRank is a MapReduce implementation of the well-known Google's PageRank algorithm [3]. PageRank can be implemented with an iterative algorithm and applied to a graph dataset. Assume each node p_i in the graph has a PageRank $PR(p_i)$. $M(p_i)$ represents the

[8] http://www.mathworks.com/help/techdoc/ref/lsqnonneg.html

set of neighboring nodes of p_i that have outlinks pointing to p_i. $L(p_j)$ is the total number of outlinks the node p_j has. d is the damping factor and N is the total number of nodes. The following equation calculates the PageRank for each node p_i.

$$PR(p_i) = (1-d)/N + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \tag{1.17}$$

PageRank values are updated in multiple rounds until they converge. In one round of PageRank MapReduce program, all nodes' PageRank values are updated in parallel based on the above equation. Concretely, the Map program distributes a share of each node's PageRank, i.e., $PR(p_j)/L(p_j)$, to all its outlink neighbors. The Reduce program collects the shares from its neighbors and applies the equation to update the PageRank. The complexity function $g()$ is also linear to the size of the input of Reduce.

Join is a MapReduce program that joins a large file with a small file based on a designated key attribute, which mimics the Join operation in relational database. The large files are the text files randomly generated with RandomTextWriter. The small file consists of 50 randomly generated lines using the same method for generating the large text dataset. The first word of each line in both types of file serves as the join key. The Map program emits the lines of the input large and small files. Each line of the small file is labeled so that they can be distinguished from the Map output. In the Reduce, the lines are checked to find those with matched keys. If the lines from both files are found matched, a Cartesian product is applied between the two sets of lines with the same key to generate the output. Depending on the key distribution, the size of output data may vary. In the Reduce program, assume there is a λ lines are from the large file and μ lines from the small file. The result of Cartesian product is $\lambda\mu$ lines. Since $\mu \leq 50$ very small, the complexity function $g()$ is approximately linear to the input $\lambda + \mu$ lines.

1.6.3 Model Analysis

We run a set of experiments to estimate the model parameters β_i for the four programs. We randomly select the values for the three parameters M, m, and R. The number of data

chunks M is calculated by the number of selected 1GB files (one file has $1024/64 = 16$ blocks). For the in-house cluster, because all available Map slots will be used in executing the MapReduce job, we control the number of Map slots m by setting the maximum number of Map slots in the *fair scheduler*. R is randomly set to a number smaller than the total number of Reduce Slots in the system. For on-demand EC2 clusters, it is straightforward to allocate m nodes as the Map slots and R nodes for the Reduce slots.

For each tested program, we generate tens of random settings of (M, m, R). M is randomly selected from the integers $[1 \dots 150] \times 16$, i.e., the number of 1GB files $\times$ 16 blocks/file. R is randomly selected from the integers $[1 \dots 50]$. Since changing m will need to update the scheduler setting, we limit the choices of m to 30,60,90,and 120. For each setting, we record the time (seconds) used to finish the program. The examples are ordered by the time cost for further analysis.

Regression Analysis. With the transformed variables (Eq. 1.11), we can conduct a linear regression on the transformed cost model

$$T(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = \beta_0 + \sum_{i=1}^{7} \beta_i x_i. \tag{1.18}$$

Table 1.1 shows the result of regression analysis with the constraints $\beta_i \geq 0$ for programs running in the in-house cluster. R^2 is a measure for evaluating the goodness of fit in regression modeling. $R^2 = 1$ means a perfect fit, while $R^2 > 90\%$ indicates a very good fit. Note that the matlab function `lsqnonneg` also demotes the insignificant β_i and sets them to 0.

Table 1.1 shows most models have very high R^2 values, except for TableJoin on AWS. The reason of lower-quality models might be caused by either the dynamic run-time environment or the special characteristics of the program (or data) that the model does not capture. However, the TableJoin model in the local cluster shows good accuracy, which may imply the run-time environment is the main reason. The cause of the problem will be further studied in our future work.

Prediction Accuracy. We also conduct a careful analysis on the prediction accuracy of the models. The leave-one-out [6] cross validation is used to identify the average prediction

	WordCount		Sort		PageRank		TableJoin	
	Local	AWS	Local	AWS	Local	AWS	Local	AWS
β_0	51.82	0	20.55	0	25.89	37.73	47.53	3.61
β_1	28.32	54.30	0.72	21.74	12.24	10.37	12.27	20.07
β_2	0.01	0	0	0	0	0.18	0	0
β_3	9.24	0	0	0	0	0	0	14.75
β_4	0	0	4.09	3.58	6.58	0	1.60	3.01
β_5	0	0	0	0	0	26.79	0	0
β_6	0.10	0	0.59	0.05	0.51	0	0.19	0
β_7	0.38	0	0	0	0	0	0	0
R^2	**0.9751**	**0.9524**	**0.9692**	**0.9253**	**0.9847**	**0.9733**	**0.9647**	**0.8432**

Table 1.1: Results of regression analysis for the in-house cluster and AWS clusters R^2 values higher than 0.90 indicate good fit of the proposed model.

accuracy and also the outliers that have low accuracy. Concretely the leave-one-out cross validation runs in n rounds if there are n training samples. In each round, one of the n samples is used for testing, while the other $n-1$ samples for training.

Figure 1.3 and 1.4 show the comparison between the actual running time and the predicted running time for each sample case. The x-axis represents the actual running time, and the y-axis the predicted time. In ideal cases, all the points will be distributed on the line $y = x$, which is shown as the solid line. These figures show that the points are very close to the ideal line, indicating excellent prediction accuracy.

We define the average accuracy as the average relative errors (ARE) over the n rounds of testing in the cross validation. Let C_i be the real cost and $\hat{C}_i$ be the estimated cost by the trained model in the round i. We calculate ARE with the following equation.

$$ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|C_i - \hat{C}_i|}{C_i} \tag{1.19}$$

Intuitively, this represents the percentage of prediction error in terms of the actual execution time. Table 1.2 shows the AREs in leave-one-out cross validation. The result confirms most models are robust and perform well. However, certain models such as PageRank in the local cluster perform less effectively than others. A further detailed study will be performed to understand the factors affecting the modeling.

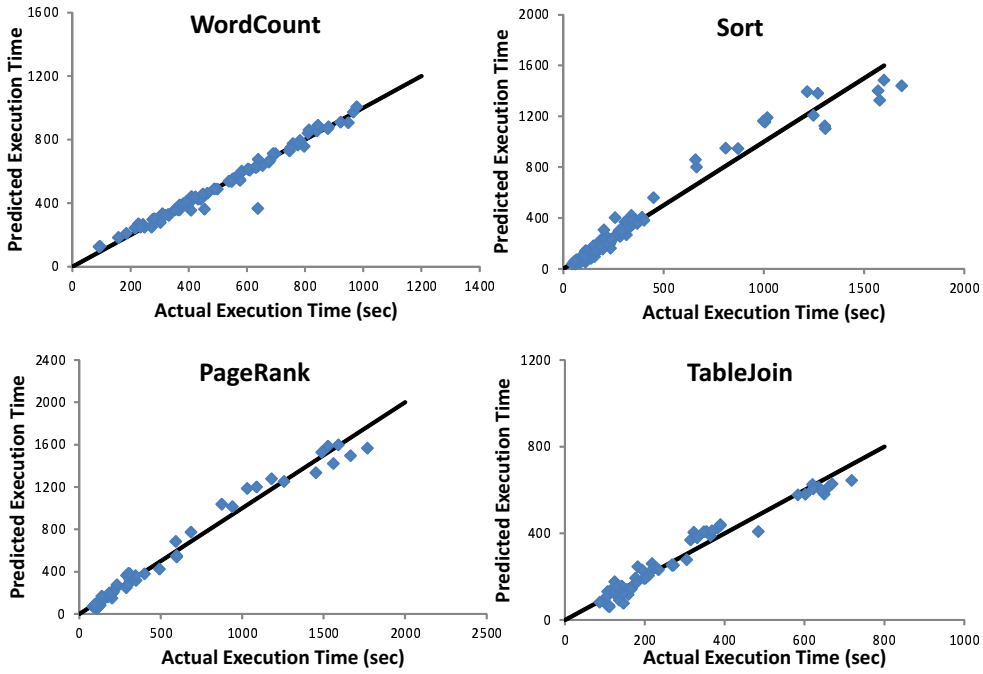

Figure 1.3: Model accuracy in local cluster.

	WordCount	Sort	PageRank	TableJoin
Local	5.49%	15.23%	12.18%	13.57%
AWS	6.46%	15.61%	7.92%	14.62%

Table 1.2: Average relative error rates of the leave-one-out cross validation and of the testing result on training data for the four programs.

1.7 Related Work

The recent research on MapReduce has been focused on understanding and improving the performance of MapReduce processing in a dedicated private Hadoop cluster. The configuration parameters of Hadoop cluster are investigated in [8, 1, 7] to find the optimal configuration for different types of job. In [21], the authors simulate the steps in MapReduce processing and explore the effect of network topology, data layout, and the application I/O characteristics to the performance. Job scheduling algorithms in the multi-user multi-job environment are also studied in [23, 19, 24]. These studies have different goals from our work, but an optimal configuration of Hadoop will reduce the amount of required resources and time for jobs running in the public cloud as well. A theoretical study on the MapReduce programming model [12] characterizes the features of mixed sequential and parallel

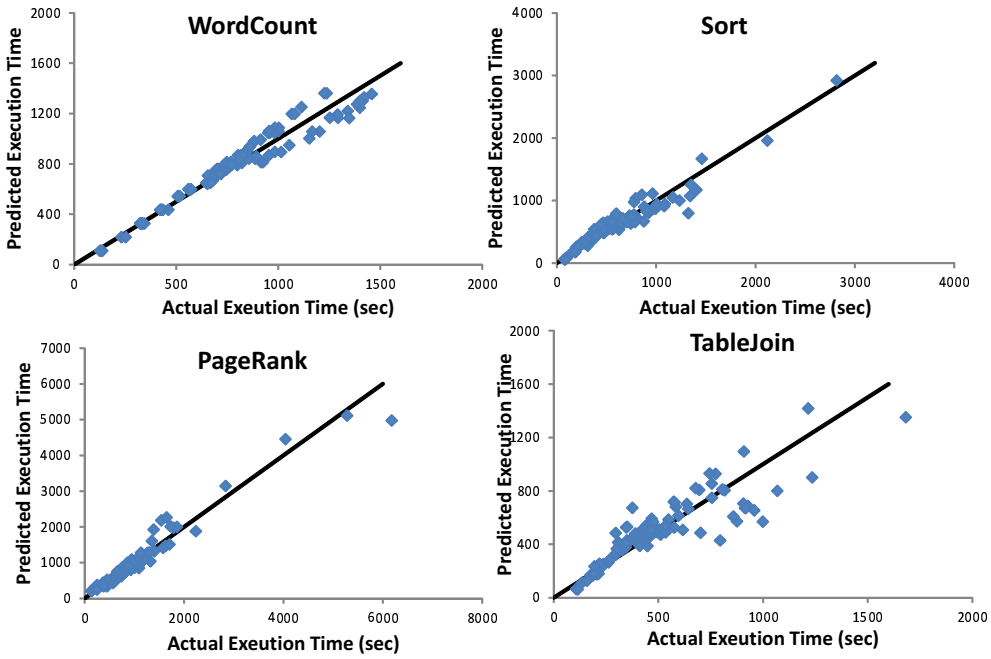

Figure 1.4: Model accuracy in Amazon EC2.

processing in MapReduce, which justifies our analysis in Section 1.3.

MapReduce performance prediction has been another important topic. Kambatla et al. [10] studied the effect of the setting of Map and Reduce slots to the performance and observed different MapReduce programs may have different CPU and I/O patterns. A fingerprint based method is used to predict the performance of a new MapReduce program based on the studied programs. Historical execution traces of MapReduce programs are also used for program profiling and performance prediction in [13]. For long MapReduce jobs, accurate progress indication is important, which is also studied in [16]. A strategy used by [10, 13] and shared by our approach is to use test runs on small scale settings to characterize the behaviors of large scale settings. However, these approaches do not study an explicit cost function that can be used in optimization problems.

1.8 Conclusion

Running MapReduce programs in the public cloud raises an important problem: how to optimize resource provisioning to minimize the financial cost for a specific job? To answer this question, we believe a fundamental problem is to understand the relationship between the amount of resources and the job characteristics (e.g., input data and processing algorithm). In this paper, we study the components in MapReduce processing and build a cost function that explicitly models the relationship between the amount of data, the available system resources (Map and Reduce slots), and the complexity of the Reduce program for the target MapReduce program. The model parameters can be learned from test runs. Based on this cost model, we can solve a number of decision problems, such as the optimal amount of resources that can minimize the financial cost with the constraints of financial budget or time deadline. We have also conducted a set of experiments on both a in-house Hadoop cluster and on-demand Hadoop clusters in Amazon EC2 to validate the model. The result shows that this cost model fits well on four tested programs. Note this modeling and optimization framework also aligns with the goal of energy efficient computing by reducing the unnecessary possession and use of cloud resources. If we can model the energy consumption profiles of the resources, we can also precisely optimize the overall energy consumption with the proposed framework.

Some future studies include (1) understand the model prediction errors to improve the modeling process, which might include sample selection and model adjustment, (2) conduct more experiments on different MapReduce programs and different types of EC2 instances, and (3) extend the study to energy efficient MapReduce computing.

Acknowledgment

This project is partly supported by the Ohio Board of Regents and Amazon Web Services.

References

[1] Shivnath Babu. Towards automatic optimization of mapreduce programs. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 137–142, New York, NY, USA, 2010. ACM.

[2] Jayant Baliga, Robert W. A. Ayre, Kerry Hinton, and Rodney S. Tucker. Green Cloud Computing: Balancing Energy in Processing, Storage and Transport. *Proceedings of the IEEE*, 99(1):149–167, jan 2011.

[3] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. In *International Conference on World Wide Web*, 1998.

[4] Abhinandan S. Das, Mayur Datar, Ashutosh Garg, and Shyam Rajaram. Google news personalization: scalable online collaborative filtering. In *International Conference on World Wide Web*, pages 271–280, New York, NY, USA, 2007. ACM.

[5] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[6] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer-Verlag, 2001.

[7] Herodotos Herodotou and Shivnath Babu. Profiling, what-if analysis, and cost-based optimization of mapreduce programs. *PVLDB*, 4(11):1111–1122, 2011.

[8] Dawei Jiang, Beng Chin Ooi, Lei Shi, and Sai Wu. The performance of mapreduce: An in-depth study. In *Proceedings of Very Large Databases Conference (VLDB)*, 2010.

[9] Thorsten Joachims, Laura Granka, Bing Pan, and Geri Gay. Accurately interpreting clickthrough data as implicit feedback. In *Proceedings of ACM SIGIR Conference*, 2005.

[10] Karthik Kambatla, Abhinav Pathak, and Himabindu Pucha. Towards optimizing hadoop provisioning in the cloud. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud09)*, 2009.

[11] U Kang, Charalampos E. Tsourakakis, and Christos Faloutsos. Pegasus: Mining peta-scale graphs. *Knowledge and Information Systems (KAIS)*, 2010.

[12] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. A model of computation for mapreduce. In *Symposium on Discrete Algorithms (SODA) (2010)*, 2010.

[13] Soila Kavulya, Jiaqi Tan, Rajeev Gandhi, and Priya Narasimhan. An analysis of traces from a production mapreduce cluster. In *IEEE/ACM International Conference on Cluster Cloud and Grid Computing*, pages 94–103, 2010.

[14] Charles L Lawson and Richard J Hanson. *Solving Least Squares Problems*. Society for Industrial Mathematics, 1987.

[15] Jimmy Lin and Chris Dyer. *Data-intensive text processing with MapReduce*. Morgan and Claypool Publishers, 2010.

[16] Kristi Morton, Abram Friesen, Magdalena Balazinska, and Dan Grossman. Estimating the progress of mapreduce pipelines. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2010.

[17] Biswanath Panda, Joshua S. Herbach, Sugato Basu, and Roberto J. Bayardo. Planet: Massively parall learning of tree ensembles with mapreduce. In *Proceedings of Very Large Databases Conference (VLDB)*, 2009.

[18] 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 ACM SIGMOD Conference*, 2009.

[19] Thomas Sandholm and Kevin Lai. Mapreduce optimization using regulated dynamic prioritization. In *SIGMETRICS/Performance09*, 2009.

[20] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *Proceedings of ACM SIGMOD Conference*, pages 1013–1020. ACM, 2010.

[21] Guanying Wang, Ali Butt, Prashant Pandey, and Karan Gupta. A simulation approach to evaluating design decisions in mapreduce setups. In *the IEEE/ACM Intl. Symposium on Modelling, Analysis and Simulation of Computer and Telecomm. Systems*, 2009.

[22] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[23] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. Job scheduling for multi-user mapreduce clusters. Technical Report UCB/EECS-2009-55, University of California at Berkeley, april 2009.

[24] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving mapreduce performance in heterogeneous environments. In *8th USENIX Symposium on Operating Systems Design and Implementation(OSDI08)*, 2008.