Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds

Fengguang Tian, Keke Chen
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: {tian.9, keke.chen}@wright.edu

***Abstract*—Running MapReduce programs in the public cloud introduces the important problem: how to optimize resource provisioning to minimize the financial charge for a specific job? In this paper, we study the whole process of MapReduce processing and build up a cost function that explicitly models the relationship between the amount of input data, the available system resources (Map and Reduce slots), and the complexity of the Reduce function for the target MapReduce job. The model parameters can be learned from test runs with a small number of nodes. 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 a time deadline or minimize the time under certain financial budget. Experimental results show that this cost model performs well on tested MapReduce programs.**

***Keywords*-MapReduce; Cloud Computing; Resource Provisioning; Performance Modeling**

I. 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 [7] and for understanding online user behaviors [17]. 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 [15], [6], [14], [9], [4], [12].

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 the public cloud has become a realistic option for most users. In view of this requirement, Amazon has developed the 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.

Running a Hadoop cluster on top of the public cloud shows different features from a private Hadoop cluster. First, for each job a dedicated Hadoop cluster will be started on a number of virtual nodes. 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. To our knowledge, there is no effective method helping the user make this decision.

The problem of optimal resource provisioning 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. It is tricky to find the best setting that minimizes the cost. With other constraints such as a time deadline or a financial budget to finish the job, this problem appears more complicated.

We propose a method to help the user make the decision of resource provisioning for running the MapReduce programs in public clouds. This method is based on the proposed MapReduce cost model that has a number of parameters to be determined for a specific application. The model parameters can be learned with tests running on a small number of virtual nodes and small test data. Based on the cost model and the estimated parameters, the user can find the optimal setting 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 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 linear model in terms of transformed variables. Linear models provide robust

[1] aws.amazon.com/elasticmapreduce/.

[2] e.g., wiki.apache.org/hadoop/AmazonEC2

generalization power that allows one to determine the parameters 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 between 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 to validate the cost model. The experimental result shows 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 2, we introduce the MapReduce Programming model and the normal setting for running Hadoop on the public cloud. In Section 3, we analyze the execution of MapReduce program and propose the cost model. In Section 4, the aforementioned decision problems on resource provisioning are formulated as several optimization problems based on the cost model. In Section 5, we present the experimental results that validate the cost model. In Section 6, the related work on MapReduce performance analysis is briefly discussed.

II. Preliminary

Although MapReduce has been a common concept in program languages for decades, MapReduce programming for large-scale parallel data processing was just recently proposed by Dean et al. in Google [5]. MapReduce is more than a programming model - it also includes the system support for processing the MapReduce jobs in parallel in a large scale cluster. A popular open source implementation of the MapReduce framework is Apache Hadoop that also includes the underlying Hadoop Distributed File System (HDFS).

It is best to understand how MapReduce programming works with an example - the WordCount program. The following code snippet shows how this MapReduce program works. 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.

When deploying a Hadoop cluster on a public cloud, we need to request a number of virtual nodes from the cloud and start them with a system image that has the Hadoop package preinstalled. In addition, the user's data may reside in the cloud storage system, e.g., Amazon S3, for which the Hadoop system needs to be appropriately configured. The configuration files are passed to the corresponding master and slave nodes, and the Hadoop cluster then gets 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 MapReduce's cost model.

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⟩);
```

III. Cost Model of 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. This cost model is the core component for solving the resource prediction and optimization problems.

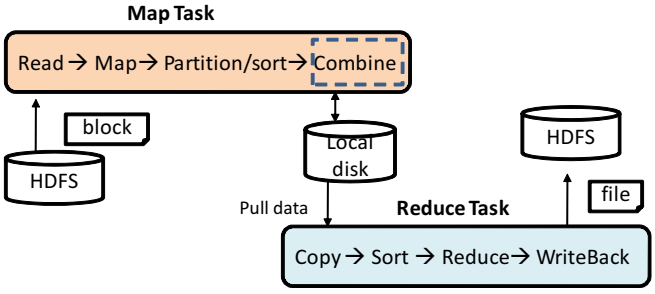

Figure 1. Components in Map and Reduce tasks and the sequence of execution.

The MapReduce processing is a mix of sequential and parallel processing. The Map phase is executed before the Reduce phase[3], as Figure 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 the

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

processes can be run in the slots in parallel 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 or Reduce slots per core. 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, which are assigned to the m slots. In the ideal case, m Map processes occupy the m slots and run in parallel - 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, say R, can be set by the user and determined by the application requirement. Similarly, if $R > r$, more than one round of Reduce processes are scheduled. In practice, to avoid the cost of scheduling multiple rounds of Reduce processes, the number of Reduce processes is often set to the same as or less than the number of Reduce slots in the cluster[4].

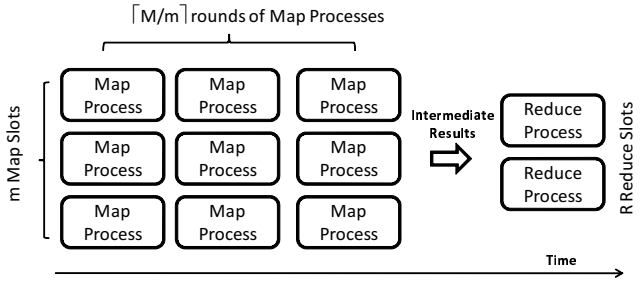

Figure 2. Illustration of parallel and sequential execution in the ideal situation.

Figure 2 illustrates the scheduling of Map and Reduce processes to the Map and Reduce slots in the ideal situation. In practice, in one round Map processes 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. We will consider the variance in cost 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 the resources and the amount of time to finish the MapReduce job.

[4]In general, the number of Reduce processes, R, is not larger than the number of Map output keys, because one Reduce process handles one or more output keys. In many applications, the number of Map output keys is so large that R is often set to the number of all available Reduce slots to optimize the performance [19].

In addition to the cost of Map and Reduce processes, the system has some additional cost managing and scheduling the M Map processes and the R Reduce processes. Based on this understanding, we analyze the cost of each Map process and Reduce process, respectively, and then derive the overall cost model.

A. 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 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 function, the complexity of which is determined by the input data size b, denoted as $f(b)$. The Map function may output data in size of $o_m(b)$ that is often a linear function to the input size b. The output will be a list of $\langle key, value \rangle$ pairs. The result will be sorted by the key and partitioned into R shares for the R Reduce processes. We denote the cost of partitioning and sorting with $s(o_m(b))$. Since the partitioning process uses a hash function to map the keys, the cost $s(o_m(b))$ is independent of R. Let's skip the Combiner component temporarily and we will discuss the situation having the Combiner component later.

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

$$\Phi_m = i(b) + f(b) + s(o_m(b)). \quad (1)$$

This cost is only related to the size of the data block b and the complexity of the Map function. It is independent of the parameters M,R and r.

B. Reduce Process

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

Assume 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

$$b_R = M \cdot o_m(b) \cdot k/R. \quad (2)$$

The Copy cost is linear to b_R, denoted as $c(b_R)$. A Merge-Sort follows to merge the M shares from the Map results while keeping the records sorted, which has the complexity $O(b_R \log b_R)$, denoted as $ms(b_R)$.

[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 function will process the data with some complexity $g(b_R)$ that depends on the real application. Assume the output data of the Reduce function 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,

$$\Phi_r = c(b_R) + ms(b_R) + g(b_R) + wr(o_r(b_R)), \quad (3)$$

C. Putting All Together

According to the parallel execution model we described in Figure 2, the overall time complexity T depends on the number of Map rounds and Reduce rounds. By including the cost of managing and scheduling the Map and Reduce processes $\Theta(M, R)$, which is assumed to be linear to M and R, we represent the overall cost as

$$T = \lceil \frac{M}{m} \rceil \Phi_m + \lceil \frac{R}{r} \rceil \Phi_r + \Theta(M, R). \quad (4)$$

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. If we use a fixed block size b in the analysis, the cost of each Map process, Φ_m, is fixed. The cost of each Reduce process, Φ_r, is subject to the factor M and R. Since the user setting R is often the same as or less than the number of Reduce slots, r, we let $\lceil R/r \rceil = 1$. 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 2 and 3 and keeping only the variables M, R, and m in the cost model, we get the detailed model

$$\begin{aligned} &T_1(M, m, R) = \\ &\beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{M}{R} + \beta_3 \frac{M}{R} \log(\frac{M}{R}) \\ &+ g(\frac{M}{R}) + \beta_4 M + \beta_5 R + \epsilon, \end{aligned} \quad (5)$$

where β_i are the parameters describing the constant factors. $T_1(M, m, R)$ is not linear to its variables, but it is linear to the transformed components: M/m, M/R, $\frac{M}{R}\log(\frac{M}{R})$, $g(M/R)$, M, and R. The parameter β_i defines the contribution of each components in the model. Concretely, β_1 represents the fixed Map cost Φ_m; β_2 represents the parameter associated with the cost of Copy and WriteBack in the Reduce phase; β_3 represents the parameter associated with the MergeSort component in the Reduce phase; β_4 and β_5 represent the parameters for the cost associated with the management cost $\Theta()$, i.e., we assume the cost is linearly associated with the number of Map and Reduce slots: $\Theta(M, R) = \beta_4 M + \beta_5 R$; β_0 represents some constant, and ϵ represents the noise component that covers the unknown or unmodeled factors in the system. We leave the discussion on the item $g(M/R)$ later.

The simplicity of the linear model has several advantages. If this model is valid, it will allow us to robustly estimate the time complexity of larger data (i.e., larger M) and more resources (larger m and R) based on the model parameters estimated with the small settings of M, m, and R. It can also reduce the complexity of solving the related optimization problems.

With Combiner. In the Map process, the Combiner function is used to aggregate the results by the key. If there are k keys in the Map output, the Combiner function reduces the Map result to k records. The cost of Combiner is only subject to the output of the Map function. 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

$$b_R = Mk/R. \quad (6)$$

Since the important factors M and R are still there, the cost model (Equation 5) applies without any change.

Function g(). The complexity of Reduce function has to be estimated with the given application. There are some special cases that the $g()$ item can be removed from Equation 5. If $g()$ is linear to the size of the input data, then its contribution can be merged to the factor β_2, because $g(M/R) \sim M/R$. Similarly, if its complexity is $O(\frac{M}{R}\log(\frac{M}{R}))$, its contribution can be merged to β_3. In these two special cases, the cost model is simplified to

$$\begin{aligned} &T_2(M, m, R) = \\ &\beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{M}{R} + \beta_3 \frac{M}{R} \log(\frac{M}{R}) \\ &+ \beta_4 M + \beta_5 R + \epsilon, \end{aligned} \quad (7)$$

In practice, many applications can be covered by the special cases.

Observations. Let's look closer to the parameters of the simplified model T_2. First, let's fix M and R. We have $T_2 \sim 1/m$. This relationship indicates that when m is already large, the increase of m will not bring significant performance gain. In particular, if M is smaller than m, increasing m will not gain, at all. Second, let's fix M and m. Then, the function of R is more complicated, involving R, $1/R$, and $(\log R)/R$. We will have to depend on experiments to explore the function of R. Finally, if we fix m and R and increase the data size M, the complexity might be dominated by the item $\frac{M}{R}\log(\frac{M}{R})$. A Combiner function can significantly reduce the weight of this item.

IV. 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 certain limited amount of financial budget; (2) with certain time constraint; (3) and without any constraint. We formulate these problems as optimization problems based on the cost model.

In all the scenarios we consider, we assume the model parameters are determined with sample runs in small scale settings. We also assume $g()$ function is one of the two simple cases. Therefore, the simplified model T_2 is applied. Since the input data is fixed, M is constant. For simplicity, we also consider all general MapReduce system configurations [1], [6] are fixed for both small and large scale settings. With this setup, the time cost function becomes

$$T_3(m,R) = \alpha_0 + \frac{\alpha_1}{m} + \frac{\alpha_2}{R} + \frac{\alpha_3 \log R}{R} + \alpha_4 R \quad (8)$$

where

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

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). 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. 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

$$v = (m + r)/\gamma. \quad (9)$$

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

$$uvT_3(m,R) = u(m+R)T_3(m,R)/\gamma. \quad (10)$$

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

$$\begin{aligned} &minimize\ T_3(m,R) \quad (11) \\ &subject\ to\ u(m+R)T_3(m,R)/\gamma \le \phi, \\ &\quad m > 0,\ and\ R > 0. \end{aligned}$$

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

$$\begin{aligned} &minimize\ u(m+R)T_3(m,R)/\gamma \quad (12) \\ &subject\ to\ T_3(m,R) \le \tau, m > 0,\ and\ R > 0. \end{aligned}$$

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 time deadline, i.e., the constraint $T_3(m,R) \le \tau$ is removed.

Note that the T_3 model parameters might be specific for a particular type of VM instance that also 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.

With the concrete setting of the T_3 model parameters (i.e., α_i be positive or negative), these optimization problems can be convex or non-convex [2]. However, they are in the category of well-studied optimization problems - there are plenty of papers and books discussing how to solve these optimization problems. Therefore, we will skip the details of solving these problems.

V. Experiments

We design and conduct a set of experiments to validate the formulated cost model. We first give the setup of the experiments, including the experimental environment and the datasets. Four tested programs are used in experiments: WordCount, TeraSort, PageRank and Join. We then run a number of rounds of the tested programs and collect the data for regression analysis and model prediction.

A. Experimental Setup

Hardware and Hadoop Configuration. The experiments are conducted in our inhouse 16-node Hadoop cluster. Each node has two quad-core 2.3Mhz AMD Opteron 2376, 16GB memory, and two 500GB hard drives, connected with a gigabit switch. The version 0.21.0 of Hadoop is installed in the cluster. One node serves as the master node and the other as the slave nodes. The single master node runs the *JobTracker* and the *NameNode*, while each slave node run both the *TaskTracker* and the *DataNode*. Each slave node is configured with eight Map slots and six Reduce slots (about two concurrent processes per core). Each Map/Reduce process uses 400MB memory. The data block size is set to 64 MB. We use the Hadoop fair scheduler to control the total number of Map/Reduce slots available for different testing jobs.

Datasets. We use a number of generators to generate testing datasets for the benchmark programs. (1) We revised the RandomWriter tool in the Hadoop package to use a Gaussian random number generator to generate random float numbers. This data is used by the Sort program. (2) We also revised 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 dataset is used by the WordCount program and the TableJoin program. (3) The third dataset is a synthetic random graph dataset. Each line of the data 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, we will randomly choose a number of the 1GB files to simulate different sizes of input data.

B. Tested Programs.

We describe the MapReduce programs used in testing and give the complexity of each one's Reduce function, i.e., the $g()$ function. If $g()$ is linear to the input data, the simplified cost model Eq. 7 is used.

WordCount is a sample MapReduce program in the Hadoop package. The Map function splits the input text into words and the result is locally aggregated by word with a Combiner; the Reduce function 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 function, $g()$, 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 that define the key range for each Reduce. As a result, 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 function and the Reduce function 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 is an iterative algorithm applied on a graph dataset. Assume each node p_i in the graph has a PageRank $PR(p_i)$. $M(p_i)$ represents the set of neighboring nodes of p_i that have an outlink 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.

$$PR(p_i) = (1-d)/N + d \sum_{p_j \in M(p_i)} \frac{PR(p_j)}{L(p_j)} \qquad (13)$$

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 function distributes a share of each node's PageRank, i.e., $PR(p_j)/L(p_j)$, to all its outlink neighbors. The Reduce function 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 an MapReduce program that joins a large file and a small file based on a designated key, 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 function 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. If the lines from both files are found, a cartesian product is applied between the two sets of lines to generate the output. Depending on the key distribution, the size of output data may vary. In the Reduce function, 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.

C. 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). The number of Map slots m is controlled 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 each tested program, we generate 25 to 60 random settings of $< M, m, R >$. M is randomly selected from the integers $[1 \ldots 150] \times 16$, i.e., the number of 1GB files $\times$ 16 blocks/file. R is randomly selected from the integers $[1 \ldots 50]$. Since changing m will need to update the scheduler setting, we limit the choices of m to 30,60,90,and 120 - for each m. For each setting, we record the time (seconds) used to finish the program.

Regression Analysis. With $x_1 = M/m$, $x_2 = M/R$, $x_3 = \frac{M}{R}\log(\frac{M}{R})$, $x_4 = M$, and $x_5 = R$, we can conduct a linear regression on the transformed cost model

$$T(x_1, x_2, x_3, x_4, x_5) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5. \qquad (14)$$

Table I shows the result of regression analysis[6]. 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.

Figure 3, 4, 5, and 6 show the goodness of fit in a more intuitive way. To make the presentation clearer, we sort the experimental results by the time cost in an ascending order. The solid lines represent the real times observed in the

[6]We used the existing linear regression package in Matlab to fit the model. An improvement on modeling would consider more constraints such as $\beta_i \geq 0$, for $i > 0$.

	WC	Sort	PR	Join
β_0	66.66	-271.22	469.70	19.60
β_1	30.00	0.47	55.16	2.07
β_2	-0.42	-0.83	-68.08	-4.50
β_3	0.06	1.61	16.84	2.20
β_4	0.02	0.63	2.43	0.42
β_5	-0.95	5.91	1.65	-0.37
R^2	**0.9969**	**0.9689**	**0.9161**	**0.9895**

Table I
RESULT OF REGRESSION ANALYSIS. R^2 VALUES ARE ALL HIGHER THAN 0.90, INDICATING GOOD FIT OF THE PROPOSED MODEL.

experiment and the '+' marks represent the predicted times using the fitted model. The closer the two, the better quality the model has. All of the four figures show excellent fit.

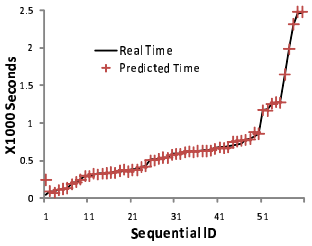

Figure 3. Fitting the model for WordCount (60 rounds).

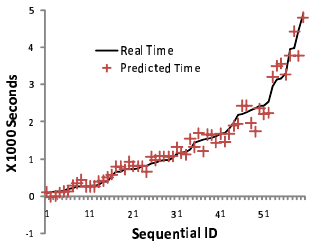

Figure 4. Fitting the model for TeraSort (60 rounds).

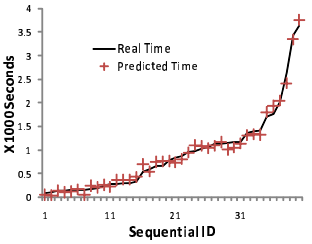

Figure 5. Fitting the model for Join program (40 rounds).

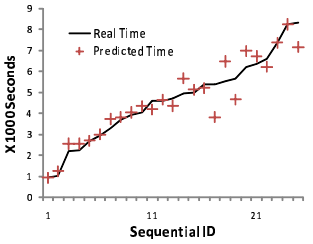

Figure 6. Fitting the model for PageRank (25 rounds).

Note that the purpose of this experiment is to show the effectiveness of the cost model. Different Hadoop clusters should result in different model parameters.

Cross Validation. We also perform a leave-one-out cross validation to study the prediction accuracy of the model. The leave-one-out cross validation runs in n rounds if there are n training examples, i.e., the tuples of (M, m, R, T). In each round, it uses one of the n examples as the testing example and the other $n-1$ examples for training the model. The accuracy is defined as the average relative errors (ARE) over the n rounds of testing. 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.

$$ARE = \frac{1}{n}\sum_{i=1}^{n}\frac{|C_i - \hat{C}_i|}{C_i} \quad (15)$$

Table II shows the relative error rates in leave-one-out cross validation. For comparison, we also list the result of testing on training data. The result confirms these models are robust and perform well.

	WC	Sort	PR	Join
Test-on-training	9.27%	16.77%	8.97%	14.82%
Leave-one-out	10.43%	18.74%	12.32%	16.83%

Table II
AVERAGE RELATIVE ERROR RATES OF THE LEAVE-ONE-OUT CROSS VALIDATION AND OF THE TESTING RESULT ON TRAINING DATA FOR THE FOUR PROGRAMS.

VI. 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 [6], [1] to find the optimal configuration for different types of job. In [18], 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 [20], [16], [21]. 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 [10] characterizes the features of mixed sequential and parallel processing in MapReduce, which justifies our analysis in Section III.

MapReduce performance prediction has been another important topic. Kambatla et al. [8] 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 [11]. For long MapReduce jobs, accurate progress indication is important, which is also studied in [13]. A strategy used by [8], [11] 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.

MapReduce has been used in handling many data intensive problems. MapReduce and Parallel databases are compared on relational data analysis jobs in [15], [6]. A few data mining algorithms have been developed based on MapReduce, including PLANET [14] for tree ensemble learning, PEGASUS [9] for mining peta-scale graphs, MapReduce EM algorithm [4], and MapReduce based text mining [12].

VII. Conclusion

Running MapReduce programs in the public cloud raises the important problem: how to optimize resource provisioning to minimize the financial cost for a specific job? 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 function for the target MapReduce program. The model parameters can be learned from test runs with small scale settings on the target program. 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 to validate the model. The result shows that this cost model fits well on four tested programs.

Due to the time limitation, we were not able to conduct experiments in the public cloud. An important ongoing work is to run experiments on Amazon EC2 nodes. The virtual machine based EC2 nodes will have different CPU, I/O, and networking characteristics from our inhouse cluster. However, since the MapReduce execution model is not changed regardless of small or large cluster, private or public cloud, we believe the effects of these system level factors can be captured by the model parameters.

Acknowledgment

We thank the anonymous reviewers for their valuable comments. This project is partly supported by Ohio Board of Regents.

References

[1] S. Babu, "Towards automatic optimization of mapreduce programs," in *Proceedings of the 1st ACM symposium on Cloud computing*. New York, NY, USA: ACM, 2010, pp. 137–142.

[2] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.

[3] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," in *International Conference on World Wide Web*, 1998.

[4] A. S. Das, M. Datar, A. Garg, and S. Rajaram, "Google news personalization: scalable online collaborative filtering," in *International Conference on World Wide Web*. New York, NY, USA: ACM, 2007, pp. 271–280.

[5] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX Symposium on Operating Systems Design and Implementation*, 2004.

[6] D. Jiang, B. C. Ooi, L. Shi, and S. Wu, "The performance of mapreduce: An in-depth study," in *Proceedings of Very Large Databases Conference (VLDB)*, 2010.

[7] T. Joachims, L. Granka, B. Pan, and G. Gay, "Accurately interpreting clickthrough data as implicit feedback," in *Proceedings of ACM SIGIR Conference*, 2005.

[8] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud09)*, 2009.

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

[10] H. Karloff, S. Suri, and S. Vassilvitskii, "A model of computation for mapreduce," in *Symposium on Discrete Algorithms (SODA) (2010)*, 2010.

[11] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An analysis of traces from a production mapreduce cluster," in *IEEE/ACM International Conference on Cluster Cloud and Grid Computing*, 2010, pp. 94–103.

[12] J. Lin and C. Dyer, *Data-intensive text processing with MapReduce*. Morgan & Claypool Publishers, 2010.

[13] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the progress of mapreduce pipelines," in *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, 2010.

[14] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "Planet: Massively parall learning of tree ensembles with mapreduce," in *Proceedings of Very Large Databases Conference (VLDB)*, 2009.

[15] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of ACM SIGMOD Conference*, 2009.

[16] T. Sandholm and K. Lai, "Mapreduce optimization using regulated dynamic prioritization," in *SIGMETRICS/Performance09*, 2009.

[17] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of ACM SIGMOD Conference*. ACM, 2010, pp. 1013–1020.

[18] G. Wang, A. Butt, P. Pandey, and K. 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.

[19] T. White, *Hadoop: The Definitive Guide*. O'Reilly Media, 2009.

[20] M. Zaharia, D. Borthakur, J. S. Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Job scheduling for multi-user mapreduce clusters," University of California at Berkeley, Tech. Rep. UCB/EECS-2009-55, april 2009.

[21] M. Zaharia, A. Konwinski, A. D. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *8th USENIX Symposium on Operating Systems Design and Implementation(OSDI08)*, 2008.