CRESP: Towards Optimal Resource Provisioning for MapReduce Computing in Public Clouds

Keke Chen, James Powers, Shumin Guo, Fengguang Tian
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, powers.4, guo.18, tian.9}@wright.edu

Abstract—Running MapReduce programs in the cloud introduces this unique problem: how to optimize resource provisioning to minimize the monetary cost or job finish time for a specific job? We study the whole process of MapReduce processing and build up a cost function that explicitly models the relationship among the time cost, 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. Based on this cost function, we can solve a number of decision problems, such as the optimal amount of resources that can minimize monetary cost within a job finish deadline, minimize time cost under certain monetary budget, or find the optimal trade-offs between time and monetary costs. Experimental results show that the proposed approach performs well on a number of sample MapReduce programs in both the in-house cluster and Amazon EC2. We also conducted a variance analysis on different components of the MapReduce workflow to show the possible sources of modeling error. Our optimization results shows that with the proposed approach we can save a significant amount of time and monetary costs, compared to randomly selected settings.

Index Terms—MapReduce; Cloud Computing; Resource Provisioning; Performance Modeling

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 and for understanding online user behaviors. 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 [3] (and its open source implementation Hadoop) has been widely used for processing and analyzing such large scale datasets.

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 that runs on-demand Hadoop/MapReduce clusters on top of Amazon EC2 nodes. There are also scripts[1] for users to manually setup Hadoop/MapReduce on EC2 nodes.

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 cluster owners do. Meanwhile, current virtualization techniques allow a virtual cluster to be provisioned or released in minutes. Thus, on-demand Hadoop clusters have become an appropriate choice for most users who have ad-hoc Hadoop jobs. Typically, such a on-demand cluster is created for a specific long-running job, where no multi-user or multi-job resource competition happens within the cluster[2] 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

1. wiki.apache.org/hadoop/AmazonEC2
2. We do not exclude the case that the user wants to pack more than one job into one cluster, especially for short jobs. However, in this paper, we only consider the large-scale jobs that may run for long time on a large dataset and thus a dedicated on-demand cluster is more practical.

method is needed to help the user make this decision.

The problem of optimizing resource provisioning for MapReduce programs involves two intertwined factors: the *monetary cost* of provisioning the virtual machine nodes and the *time cost* 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. It is tricky to find the best setting that minimizes the monetary cost. With other constraints such as a deadline or a monetary budget to finish the job, this problem appears more complicated.

We propose a method to help users make the decision of resource provisioning for running MapReduce programs in public clouds. This method, Cloud RESource Provisioning (CRESP) for MapReduce Programs, is based on the proposed *specialized MapReduce time 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 small scale settings, i.e., small clusters and small sample datasets. Based on the time cost model and the estimated parameters, the user can find the optimal setting for resources by solving certain optimization problems.

The CRESP 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 time cost model (function) can be represented as a weighted linear combination of a set of non-linear functions of these variables. These models provide robust generalization power that allows one to determine the weights of the functions with the data collected on small scale tests.
- Based on this time cost model, we formulate two important decision problems: minimizing time cost within monetary constraint, and minimizing monetary cost within time constraint. The resource requirement is mapped to the number of Map/Reduce slots; the monetary cost is proportional to the product of the time cost function and the acquired Map/Reduce slots. With our time 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 optimization results are times better than the randomly selected cluster configurations.

The entire paper is organized as follows. In Section 2, we analyze the execution of MapReduce program, propose the cost model, and describe the statistical methods for learning the model. In Section 3, we formulate several problems on resource provisioning as optimization problems based on the cost model. In Section 4, we present the experimental results that validate the cost model and analyze the modeling errors. The related work on MapReduce performance analysis appears in the supplementary file.

2 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. This cost model is the core component for solving the resource prediction and optimization problems.

Basic Ideas. We first give the formal setting of the modeling problem and then discuss the unique features of our approach. 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 $T = T(S, D, C)$.

Our approach is a combination of the white-box [5] and machine learning approaches [9]. It aims to gain good generalization power and modeling flexibility at the same time. There are a number of special features with this modeling method. (1) Because MapReduce programs have very different logic and time complexity, the cost functions should be different from application to application. However, they can share some general form, only differing in the setting of parameters. (2) The parameter setting of the cost function can also be affected by input data distributions. We argue that the same program, if the distribution of input dataset is changed, should be modeled separately as a different modeling task. (3) We aim to learn the cost function from small-scale examples with small amounts of resources and input data, which will still be robust for large-scale settings. (4) We expect to learn a closed-form function, which can be nicely incorporated in optimization tasks. The learning methods [4] resulting in special forms of function such as decision trees do not fit our need.

Because of these special requirements, we aim to design a modeling method that gives a closed form function with good generalization power. This method depends on an accurate analysis of the whole process of MapReduce processing framework. The resulting model will be in the form of

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

where $h_i(S,D,C)$ are possibly some non-linear transformations of the input factors S, D, and C, k is the number of such items determined by the analysis result, and β_i should be learned for a specific application. With this modeling principle in mind, the following sections conduct the modeling analysis, give a concrete formulation of the cost functions of Map task and Reduce task, which helps find the components, $h_i(S,D,C)$, and finally we will integrate these components to have the whole cost function.

2.1 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 5 shows. However, in each phase many Map or Reduce tasks are executed in parallel. To clearly describe the MapReduce execution, we would like to distinguish the concepts of *Map/Reduce slot* and *Map/Reduce task*. Each Map (or Reduce) task is executed in a Map (or Reduce) slot. A slot is a unit of computing resources allocated for running the tasks. According to the system capacity, a computing node can only accommodate a fixed number of slots so that the tasks 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 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 task as the Map/Reduce program running on a specific slot. By default, in Hadoop each Map task handles one chunk of data (e.g., 64MB). Therefore, if there are M chunks of data, M Map tasks in total will be scheduled and assigned to the m slots. In the ideal case, m Map tasks can run in parallel in the m slots - we call it one round of Map tasks. If $M > m$, which is normal for large datasets, $\lceil M/m \rceil$ Map rounds are needed.

Different from the total number of Map tasks, the number of Reduce tasks, 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 tasks[4]. Similarly, if $R > r$, more than one round of Reduce tasks are scheduled. It is probably not very helpful to set a R greater than r because there is no restriction on the amount of data a Reduce task 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 [13]. 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 monetary cost.

Figure 6 (in Supplementary Section 7.1) illustrates the scheduling of Map and Reduce tasks to the Map and Reduce slots in the ideal situation. In practice, Map tasks 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, disk I/O, network traffic, and 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, the resources are provisioned at cost. There is a tradeoff between the amount of resources and the amount of time to finish the MapReduce job.

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

2.2 Cost of Map Task

A Map task can be divided into a number of sequential components, including Read, Map, Sort/Partition, and optionally Combine, as Figure 5 (in Supplementary Section 7.1)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 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 tasks. 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

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

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

skip the Combiner component temporarily and we will revisit the Combiner component later.

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

$$\Phi_m = i(b) + f(b) + s(o_m(b), R) + \epsilon_m. \quad (2)$$

$i(b)$ and $f(b)$ is only related to the size of the data block b and the complexity of the Map function, 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

$$\Phi_m(m, M, r, R) = \mu_1 + \mu_2 R + \epsilon_m, \quad (3)$$

where μ_1, μ_2, and the distribution of ϵ_m are constants specific to each application. We will study the variance of ϵ_m to understand the modeling accuracy.

2.3 Cost of Reduce Task

A Reduce task has these components: Shuffle, Merge-Sort, Reduce and WriteResult. They are also sequentially executed in the Reduce task.

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

$$b_R = M \cdot o_m(b) \cdot k/R. \quad (4)$$

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 Shuffle 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 $b_R \lceil \log_B M \rceil$.

The Reduce function will process the data with some complexity $g(b_R)$ that depends on the specific 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 Reduce task is the sum of the component costs,

$$\Phi_r = c(b_R) + ms(b_R) + g(b_R) + wr(o_r(b_R)) + \epsilon_r, \quad (5)$$

Both the Shuffle 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

$$\begin{aligned}\Phi_r(m, M, r, R) &= \lambda_1(m/R) + \lambda_2(M \log M/R) + \\ &\quad g(M/R) + \lambda_3(M/R) + \epsilon_r, \quad (6)\end{aligned}$$

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

2.4 Putting All Together

According to the parallel execution model we described in Figure 6, 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 tasks $\Theta(M, R) = \xi_1 M + \xi_2 R$ is linear to M and R, as stated in the documentation [13]. By including all costs, we represent the overall cost as

$$T = \lceil \frac{M}{m} \rceil \Phi_m + \lceil \frac{R}{r} \rceil \Phi_r + \Theta(M, R). \quad (7)$$

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 tasks 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 $R = r$. 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 3 and 6 and keeping only the variables M, R, and m in the cost model, we get the detailed model

$$\begin{aligned}&T_1(M, m, R) = \\ &\beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{MR}{m} + \beta_3 \frac{m}{R} + \beta_4 \frac{M \log M}{R} \\ &+ \beta_5 M/R + \beta_6 M + \beta_7 R + \beta_8 g(\frac{M}{R}) + \epsilon, \quad (8)\end{aligned}$$

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

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

are the weights of each components derived in the component-wise cost analysis. According to the meaning of the components, β_i cannot be negative values. Finally, ϵ represents the overall noise, which is the combination of noises from the two phases and other global factors. We leave the discussion on the item $g(M/R)$ later.

With Combiner. In the Map task, 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 task and thus affects the cost of the Reduce phase. With the Combiner, the amount of data that a Reduce task needs to pull from the Map is changed to

$$b_R = Mk/R. \tag{9}$$

Since the item M/R is still there, the cost model (Equation 8) applies without any change.

Function g(). The complexity of Reduce function has to be estimated with the specific application. There are some special cases that the $g()$ item can be removed from Equation 8. If $g()$ is linear to the size of the input data, then its contribution can be merged to the factor β_5, 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

$$T_2(M, m, R) = \beta_0 + \beta_1 \frac{M}{m} + \beta_2 \frac{MR}{m} + \beta_3 \frac{m}{R} + \beta_4 \frac{M \log M}{R} + \beta_5 M/R + \beta_6 M + \beta_7 R + \epsilon, \tag{10}$$

2.5 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 because of data distributions, specific I/O patterns, and data processing logic. 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 tasks 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 [9] is applied to learn the model from the training data with the transformed variables

$$x_1 = M/m, x_2 = MR/m, x_3 = m/R,$$
$$x_4 = (M \log M)/R, x_5 = M/R, x_6 = M, x_7 = R. \tag{11}$$

Because β_i are the weights of the components in the total cost, we have the contraints $\beta_i \geq 0$, $i = 0 \ldots r$, which require nonnegative linear regression [4] to solve the learning problem. The cross-validation method [4] is then used to validate the performance of the learned model. We will show more details in experiments.

3 Optimizing 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 monetary 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. 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 [2], [6], [5] and fixed for both small and large scale settings. With this setup, the time cost function becomes

$$T_3(m, R) = \alpha_0 + \frac{\alpha_1}{m} + \frac{\alpha_2 R}{m} + \frac{\alpha_3 m}{R} + \frac{\alpha_4}{R} + \alpha_5 R \tag{12}$$

where

$$\alpha_0 = \beta_0 + \beta_6 M, \alpha_1 = \beta_1 M, \alpha_2 = \beta_2 M,$$
$$\alpha_3 = \beta_3, \alpha_4 = \beta_4 M \log M + \beta_5 M, \alpha_5 = \beta_7.$$

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 that homogenous VM instances are used in the deployment. According to the capacity of a virtual machine (CPU cores, memory, disk and network bandwidth), a VM instance 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 a specific cluster setup. 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

$$v = \lceil (m + r)/\gamma \rceil. \tag{13}$$

If the price of renting one VM instance for an hour is u and T_3 returns time cost in seconds, the total monetary cost is determined by the result $uv\lceil T_3(m, R)/3600 \rceil$. Since we usually set R to r, it follows that the total monetary cost for renting the Hadoop cluster is

$$uv\lceil T_3(m, R) \rceil = u\lceil T_3(m, R) \rceil \lceil (m + R)/\gamma \rceil. \tag{14}$$

Now we are ready to formulate the optimization problems.

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

$$\begin{aligned} &\mathit{minimize}\ T_3(m, R) \qquad (15)\\ &\mathit{subject\ to}\ u\lceil T_3(m, R)\rceil \lceil (m + R)/\gamma\rceil \le \phi,\\ &\quad m > 0,\ \mathit{and}\ R > 0. \end{aligned}$$

2) If the constraint is about the maximum amount of time τ for finishing the job, the problem of minimizing the monetary cost can be formulated as

$$\begin{aligned} &\mathit{minimize}\ u\lceil T_3(m, R)\rceil \lceil (m + R)/\gamma\rceil \qquad (16)\\ &\mathit{subject\ to}\ T_3(m, R) \le \tau, m > 0,\ \mathit{and}\ R > 0. \end{aligned}$$

3) The second 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 constraint $T_3(m, R) \le \tau$.

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.

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. In practice, since we often look at the *integer* parameters m and R in a limited range, e.g., [1..1000], a brute-force search can be applied to find the optimal solution by simply enumerating all the combinations of m and R.

Impact of Modeling Error. The learned model is normally not perfect, resulting in prediction error. As the optimization formulation uses the model to formulate the time and monetary costs, we should understand how these errors will affect optimization. When the error is introduced, the optimization objectives are actually minimizing the average time or monetary cost, which should still be consistent with the actual optimization objectives. However, the time and monetary constraints are now set to their average values, which might not be satisfactory. In the following, we formally define the prediction error and then describe the methods for computing the relaxed constraints.

Let C be the real execution time and $\hat{C}$ be the predicted. Let δ be the model prediction error rate, which defined as the relative error to the real execution time, i.e., $\delta = |\hat{C} - C|/C$. Without loss of generality, we assume $0 < \delta < 1$ and δ can be estimated with the training examples (as described in Experiments). Let a be a positive constant, $a >= 1$, we expect the actual error rate will satisfy $a\delta > |\hat{C} - C|/C$. Thus, we can derive that with certain probability the actual time cost C is within the following range

$$\hat{C}/(1 + a\delta) \le C \le \hat{C}/(1 - a\delta).$$

In the worst case the time cost will be around $\hat{C}/(1 - a\delta)$. Because the monetary cost is represented as $u\lceil C\rceil \lceil (m + R)/\gamma\rceil$, correspondingly the worst case financial charge will be $u\lceil \hat{C}/(1 - a\delta)\rceil \lceil (m + R)/\gamma\rceil$. The constraints can be adjusted to these worst-case formulation. For example, for the time constraint $T_3(m, R) \le \tau$, we may use $T_3(m, R)/(1 - a\delta) \le \tau$ instead.

4 Experiments

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, Sort, PageRank and TableJoin, which are used in evaluating the cost model. Next, a comprehensive evaluation on both the in-house cluster and Amazon Cloud will be conducted to show the model goodness of fit and the prediction accuracy. Then, we present a in-depth variance analysis on the Map and Reduce phases to understand the potential modeling errors. Finally, we show that with the models we learned and the optimization method we discussed, we can help users achieve optimal resource provisioning schemes.

4.1 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 task 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

6. http://hadoop.apache.org/docs/r1.1.1/fair_scheduler.html

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 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.

Sample Programs. We will use four sample MapReduce program in our evaluation: WordCount, Sort, PageRank, and TableJoin (details in Supplementary Section 7.2)

4.2 Model Fitting

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, we allow each node to have only one slot. It is thus straightforward to allocate m nodes as the Map slots and R nodes for the Reduce slots. For each tested program, we generate 30-60 random settings of (M, m, R) for the local cluster and 60-100 settings for EC2 clusters. For each setting, we record the time (seconds) used to finish the program.

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

$$T(x_1, x_2, x_3, x_4, x_5, x_6, x_7) = \beta_0 + \sum_{i=1}^{7} \beta_i x_i. \qquad (17)$$

Table 6 shows the result of regression analysis with the constraints $\beta_i \geq 0$ for programs running in the in-house cluster and in EC2 clusters (denoted as AWS). 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 most of insignificant β_i to 0.

Table 6 in Supplementary (Section 7.3) 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 cannot capture. The cause of the modeling error will be discussed later.

Prediction Accuracy. We also conduct a careful analysis on the prediction accuracy of the models. The leave-one-out [4] cross validation is used to identify the average prediction accuracy and possible outlier examples. 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 are for training.

The detailed comparison on the predicted time and the real time cost is shown in (Section 7.4). Below we give the overall 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.

$$ARE = \frac{1}{n} \sum_{i=1}^{n} \frac{|C_i - \hat{C}_i|}{C_i} \qquad (18)$$

Intuitively, this represents the percentage of prediction error in terms of the actual execution time. Table 1 shows the AREs in leave-one-out cross validation. The result confirms that most models are robust and perform well. However, certain models such as PageRank in the local cluster and TableJoin in both local and AWS perform less accurately than others. A further study on the component-wise variances will be performed to understand the factors affecting the modeling accuracy.

7. wiki.apache.org/hadoop/AmazonEC2
8. http://www.mathworks.com/help/techdoc/ref/lsqnonneg.html

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

TABLE 1
Average relative error rates (ARE) from leave-one-out cross validation.

Relaxed Bounds. Note that in Section 3 we have given a formal analysis on the impact of modeling error. We have developed the method to make the actual time cost within a relaxed bound based on the predicted time cost. Specifically, with the parameter δ (estimated with ARE, i.e., $\hat{\delta} = ARE$), we can expect the actual time cost C is bounded by $\hat{C}/(1-a\hat{\delta})$, $a \geq 1$, with high probability. With the leave-one-out testing, we can get the ARE, C_i, and $\hat{C}_i$ and then we can identify the proportion of in-bound examples. Table 2 gives the in-bound rates with $a = 1$ and $a = 2$, respectively. The results show that with $a = 1$ the in-bound rates are already very high, while with $a = 2$ the rates are 90 % $\sim$ 100%.

	WordCount	Sort	PageRank	TableJoin
Local (a=1)	92.5%	77.1%	77.1%	86.6%
AWS (a=1)	75.0%	90.7%	77.89%	82.56%
Local (a=2)	96.3%	89.6%	91.7%	95.8%
AWS (a=2)	89.0%	100%	89.5%	100%

TABLE 2
The percentage of examples that have real time costs within the relaxed bounds.

Predicting Time Costs of Large Settings. Due to the economics reason, it is inappropriate to use very large datasets for training that have a comparable size to the original data, e.g., $> 1/10$ of the original size. Ideally the modeling data will be much small, in a range of tens of gigabytes at most. In previous experiments, we randomly select datasets that have sizes ranging from a few gigabytes to near 100 gigabytes. To understand how the small setting examples that have smaller input data work in modeling, we sort the training examples by their input data size. The smaller setting examples are used for training and the larger examples for testing.

Figure 1 and 2 show the results for the local hadoop cluster and the AWS clusters, respectively. The x-axis indicates that the examples with input data sizes $<= x$ GB are used for training and the rest of the examples for testing. The y-axis represents the testing error, i.e., the ARE values. For the local cluster, curves are kind of flat, except for PageRank, which means the small setting examples work pretty well for generating good global models.

For AWS clusters, the results show different patterns. For training data $<= 10GB$, the learned models perform excellent on large settings. With more training examples in (10GB, 20GB] included, except for Sort's model, all other models still perform very well. Interestingly, further including training examples in (20GB, 60GB] will significantly reduce the model quality, except for the PageRank models. While including addtional examples with $>$ 70GB, the model quality recovers soon. It indicates that the examples from the range (20GB, 60GB] are probably very noisy. These noises can be possibly identified and removed with the methods discussed in the literature [9], but the reason of generating the noisy data will be investigated. Compared to the results from the local cluster, the AWS results show larger variances, which might be caused by the multi-tenant virtualized environment.

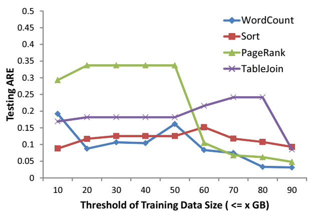


Fig. 1. Accuracy of predicting large setting samples for the local cluster.

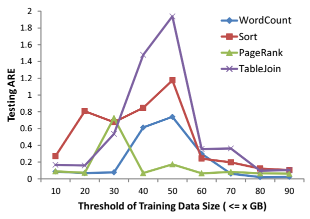


Fig. 2. Accuracy of predicting large setting samples for AWS clusters.

Cost of Experiments. One may wonder how much money and time it will need to get the modeling done. We evaluate the costs of the experiments we have done with AWS so far and list the results in Table 3. Overall, the AWS charge is around \$38-\$77, while the time cost is about 12-27 hours. For processing large scale data, we expect the amount of data will be 10-100 times over the sizes we use in modeling. Thus, the monetary and time costs of modeling will be quite acceptable.

	WordCount	Sort	PageRank	TableJoin
# of examples	100	76	96	87
Money (\$)	68	48	76	38
Time (hours)	22	15	27	12

TABLE 3
Total costs of AWS experiments.

Note that optimizing these costs is not in the scope of our current study, which, however, will be an important task in our ongoing work. The future research on reducing modeling costs will be conducted in two directions: (1) developing methods for designing training examples, i.e., selecting the most representative parameter settings to generate training examples, and (2) studying the minimum number of training examples that are is sufficient to derive high-quality models.

4.3 Optimizing Resource Provisioning

We have developed a simple optimization algorithm based on the brute force method discussed in Section 3 . It takes the model parameters β_i, the maximum job time τ (or the monetary budget ϕ), the cluster configuration: γ slots per node, and the per-node price u as the input, and outputs the optimal setting m and

R that minimizes the monetary cost (or the time cost) . This program will be open-sourced.

With this program, we compare the optimization results on 64GB input data with sample jobs running in EC2 clusters. We try to simulate the decision process for users who have no knowledge about parameter selection. Due to the EC2 quota we have, we limit our selection of m and R in [1,400]. The user's decision on parameter selection is defined as three types: conservative [1,100], neutral [101, 300], and aggressive [301, 400]. We randomly generate 50 parameter settings for each of the three ranges and record the jobs' time and monetary costs. Mid-size instances are used for experiments, where γ is set to four slots, and the price per instance hour is \$0.12.

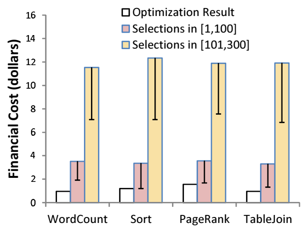


Fig. 3. Monetary costs of the optimal setting and randomly selected settings, time constraint=0.5 hour.

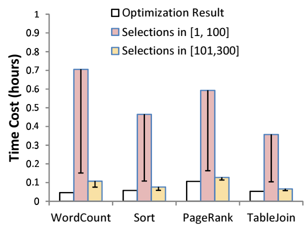


Fig. 4. Time costs of the optimal setting and randomly selected settings, budget constraint=\$10.

(1) The first set of experiments uses the time constraint $\tau = 0.5$ hour, and we want to find the setting that minimizes the monetary cost. Figure 3 shows the optimal cost given by our optimization algorithm and the average cost of the randomly generated settings in the ranges [1, 100] and [101, 300] respectively. The lower end of the error bar shows the minimum cost in the randomly selected settings. The random settings in [301, 400] are much more expensive than other settings and thus not included in the figure. Overall, the average dollar costs in [1, 100] are still significantly higher than the dollar costs of the optimization results, although the best result might be close to the optimal result. Let's look more details on the range [1, 100].

Table 4 shows the comparison between the random settings in [1, 100] and the optimization result. "Average/Opt" represents the rate between the average of random settings and our optimization result. Note that the number of settings in [1, 100] missing the time constraint is also significant. When the range is extended to [101, 300] and [301, 400], with more resources the time constraint is 100% satisfied for all random settings, but the costs are too high to be comparable.

(2) The second set of experiments uses the money budget $\phi = \$10$, and we want to find the setting that minimizes the time cost. Figure 4 shows the optimal time cost given by our optimization algorithm and the average time cost of the randomly generated settings in range [1, 100] and [101, 300]. All random settings in [301, 400] do not satisfy the money budget and thus excluded by the figure. The selections in [101, 300] have closer results to the optimization one than those in [1, 100]; however, they also have much higher constraint missing rates ($> 70\%$), which are risky to try. Overall, the optimization results are still significantly better than the averages of random selections. Similarly, we list the concrete comparison in Table 5 .

	WordCount	Sort	PageRank	TableJoin
Average/Opt	3.66	2.80	2.28	3.44
RandomBest/Opt	2.0	1.0	1.08	1.38
Missing Rate	24%	20%	38%	18%

TABLE 4
Comparison on monetary costs between random settings in [1, 100] and the optimization.

	WordCount	Sort	PageRank	TableJoin
Average/Opt	2.34	1.31	1.20	1.25
RandomBest/Opt	1.65	1.01	1.08	1.08
Missing Rate	76%	80%	78%	78%

TABLE 5
Comparison on time costs between random settings in [101, 300] and the optimization result.

5 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, such as [6], [2], [5], [12], [7]. 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.

The Starfish project [5] tries to model the data flow of the whole MapReduce program in great detail. The model depends on low level parameters such as Map output selectivity, and spill buffer pairs, etc., that are potentially affected by the Hadoop system setting, the input data, and the Map/Reduce functions. Because the parameter space is large, they use a subspace random enumeration method to find the approximate optimal setting. This approach differs from ours in at least two aspects. (1) The optimization goal is different. Starfish aims to find the optimal setting of the Hadoop system parameters for a fixed cluster that minimizes the execution time of specific jobs, while our approach assumes the system parameters are already set but the scale of cluster is to be determined. (2) The modeling method may result in large estimation errors, as it excludes the performance variances and the cost of Map and Reduce functions. However, it is not important for their modeling purpose as long as the model's predict time costs are closely correlated with the real costs, so that both the model and the real cost function will reach the minimum around the same parameter setting. In contrast, our approach needs accurate estimation to have the constraints satisfied.

The RoPE approach [1] aims to optimize complex relational query execution plans that consist of multiple MapReduce programs. Due to the changing data distributions, e.g., reduce-phase data selectivity, a fixed cost model does not work well and the execution plan needs to be dynamically adjusted to optimize the resource configuration and minimize execution time. One important idea is to use run-time data statistics to help decision making on the fly. However, only heuristic rules or algorithms in traditional relational query optimization are used in this approach, which may improve the resource configuration compared to the static execution plans, but may not reach the optimal or even near-optimal solutions. In contrast, our approach uses sample runs to learn the model parameters. Starting a cluster and then dynamically adjusting the configuration is not very efficient for our application context.

Verma et al. [11] also considered the resource provisioning problem for running MapReduce programs in public clouds. In addition, they take system failures into consideration and derive the theoretical bounds when failure happens. They characterize a specific job with a set of "job performance invariants", a similar idea to the β parameters in our modeling. However, the modeling approach is entirely different and seemingly rough in a sense that the cost is modeled approximately linear to the amount of input data. The failure modeling is theoretically valuable, but the derived bounds are far from the possible cost variances brought by data locality and system performance fluctuation. It would be more valuable to model these variances as our approach does.

6 Conclusion

Running MapReduce programs in the public cloud raises an important problem: how to optimize resource provisioning to minimize the monetary 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 function 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 monetary cost with the constraint on monetary budget or job finish time. 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 approach. The result shows that this cost model fits well on four sample programs. We have also conducted a in-depth error analysis to show the sources of potential modeling errors. The results show that by using our optimization approach we can save about 80% in either time cost or monetary cost.

We plan to do some studies in the future. (1) We will study the sample selection and model learning method to further improve the model quality and reduce the cost of collecting training examples; (2) we will conduct more experiments on different MapReduce programs and on different types of EC2 instances; and (3) we will incorporate existing studies on reducer skew balancing to understand the effectiveness of our modeling method for the rebalanced MapReduce processing.

Acknowledgment

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

References

[1] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou, "Re-optimizing data-parallel computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 21–21.

[2] 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.

[3] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *OSDI*, 2004, pp. 137–150.

[4] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*. Springer-Verlag, 2001.

[5] H. Herodotou and S. Babu, "Profiling, what-if analysis, and cost-based optimization of mapreduce programs," *PVLDB*, vol. 4, no. 11, pp. 1111–1122, 2011.

[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] 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.

[8] Y. Kwon, M. Balazinska, B. Howe, and J. Rolia, "Skewtune: mitigating skew in mapreduce applications," in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '12. New York, NY, USA: ACM, 2012, pp. 25–36.

[9] J. Neter, M. Kutner, C. Nachtsheim, and W. Wasserman, *Applied Linear Statistical Models, 3rd Ed.* WCB/McGraw-Hill, 1996.

[10] S. R. Ramakrishnan, G. Swart, and A. Urmanov, "Balancing reducer skew in mapreduce workloads using progressive sampling," in *Proceedings of the Third ACM Symposium on Cloud Computing*, ser. SoCC '12. New York, NY, USA: ACM, 2012, pp. 16:1–16:14.

[11] A. Verma, L. Cherkasova, and R. H. Campbell, "Resource provisioning framework for mapreduce jobs with performance goals." in *Middleware Conference*, 2011, pp. 165–186.

[12] 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.

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

Keke Chen is an assistant professor in the Department of Computer Science and Engineering, and a member of the Ohio Center of Excellence in Knowledge Enabled Computing (the Kno.e.sis Center), at Wright State University. He directs the Data Intensive Analysis and Computing (DIAC) Lab at the Kno.e.sis Center. He earned his PhD degree from Georgia Institute of Technology in 2006, his Master's degree from Zhejiang University in China in 1999, and his Bachelor's degree from Tongji University in China in 1996. All degrees are in Computer Science. His current research areas include visual exploration of big data, secure data services and mining of outsourced data, privacy of social computing, and cloud computing. During 2006-2008, he was a senior research scientist at Yahoo! Labs, working on web search ranking, cross-domain ranking, and web-scale data mining. He owns three patents for his work in Yahoo!.

James Powers has earned BS and MS degrees in Computer Science and is currently a PhD candidate at Wright State University. He is a member of the Data Intensive Analysis and Computing (DIAC) Lab. His research areas include big data analysis with an emphasis on practical applications of homomorphic encryption for confidential computations on big data in a public cloud. He has nearly thirty years of experience in the design, development, and management of large and small scale computer systems in both commercial and government settings. He holds certifications in a variety of technologies including Teradata, Oracle, Linux, and Security +. He was awarded the Department of Defense Superior Management Award by the Assistant Secretary of Defense for his development of an in-transit visibility system in support of Operation Desert Storm.

Shumin Guo is currently a PhD student in the Department of Computer Science and Engineering, and a member of the Data Intensive Analysis and Computing (DIAC) Lab, at Wright State University, Dayton, OH, USA. He received his Master's degree in Electronics Engineering from Xidian University, Xi'an China, in 2008. His current research interests are privacy preserving data mining, social network analysis, and cloud computing.

Fengguang Tian earned his Master's degree in Computer Science from Wright State University in 2011, and Bachelor's degree in Electronic Engineering from Beijing University of Posts and Telecommunications, China. Currently, he is a software engineer at IBM.

7 Supplementary Materials

7.1 Diagrams for MapReduce Processing

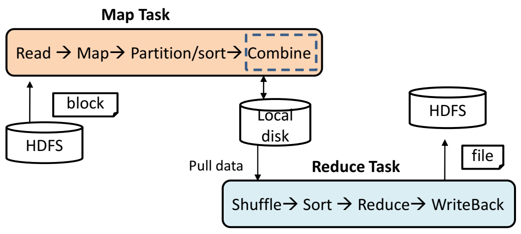

Map Task
Read → Map→ Partition/sort→ Combine
block
HDFS
Local disk
HDFS
file
Pull data
Reduce Task
Shuffle→ Sort → Reduce→ WriteBack

Fig. 5. Components in Map and Reduce tasks and the sequence of execution.

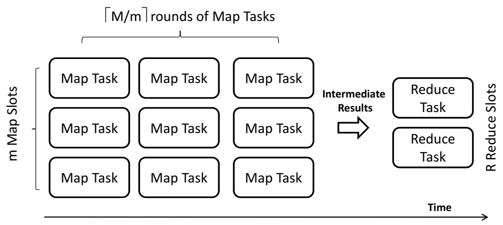


Fig. 6. Illustration of parallel and sequential execution in the ideal situation.

7.2 Sample Programs for Testing.

In this section, 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 in one of the two special cases, the simplified cost model Eq. 10 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 to define the key range for each Reduce task. 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. PageRank can be implemented with an iterative algorithm and applied to a graph dataset. 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 PageRank formula. Concretely, the Map function distributes a share of each node's PageRank to all its outlink neighbors. The Reduce function collects the shares from its neighbors and applies the PageRank formula to update the PageRank. The Reduce complexity function $g()$ is also linear to the size of the input of Reduce.

TableJoin 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 function emits the lines of the large and small input 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 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.

7.3 Details of Regression Analysis Result

Table 6 has the details for each regression model and the goodness-of-fit (R^2). $R^2 > 0.9$ indicates very good models.

	WordCount		Sort		PageRank		TableJoin	
	Local	AWS	Local	AWS	Local	AWS	Local	AWS
β_0	51.8	0	20.6	0	25.9	37.7	47.5	3.6
β_1	28.3	54.3	0.7	21.7	12.2	10.4	12.3	20.1
β_2	0	0	0	0	0	0.2	0	0
β_3	9.2	0	0	0	0	0	0	14.8
β_4	0	0	4.1	3.6	6.6	0	1.6	3.0
β_5	0	0	0	0	0	26.8	0	0
β_6	0.1	0	0.6	0.1	0.5	0	0.2	0
β_7	0.4	0	0	0	0	0	0	0
R^2	**0.98**	**0.95**	**0.97**	**0.93**	**0.98**	**0.97**	**0.96**	**0.84**

TABLE 6
Result of regression analysis for the in-house cluster and EC2 clusters. R^2 values higher than 0.90 indicate good fit of the proposed model.

7.4 Details in Prediction Error Evaluation

Figure 7 shows the comparison between the actual time cost and the predicted time cost for each sample program running in in-house cluster and EC2 clusters, respectively. The x-axis represents the actual running time, and the y-axis the predicted time. With an ideal model, all the points will be on the line $y = x$, which is the solid line in the figures. These figures show that the points are very close to the ideal line, indicating excellent prediction accuracy.

7.5 Variance Analysis

The modeling process has included the error components ϵ_m for the Map phase and ϵ_r for the Reduce phase. Each task may have variant processing time, due to the locality of data, the amount of data, the network traffic, etc., which will eventually cause the modeling error. We would like to understand how signficant the variance of each component in the whole MapReduce program can be, which can be used to improve the modeling process.

Note that the variance caused by the unbalanced load of reducers is not considered in our current study. Recent studies on this problem [10], [8] has provided some possible solutions to evenly distribute reducer's workloads. These techniques can be adapted to minimize or eliminate this variance.

With this exclusion, the possible sources of errors in our study might be caused by (1) the locality of data, i.e., fetching local data and remote data will have significant different time cost; and (2) the performance fructuation of network I/O and processing power in the multi-process/multi-tenant environment.

Among the four sample programs, we believe the Sort program is the best for observing theses component-wise variances. The processing time of each component in the MapReduce pipeline is basically determined by the amount of data processed by that component. The Sort program does not reduce the data in processing: it will pass all the data through each component in the workflow. With uniformly distributed input data, we can safely exclude the effect of data skew and observe the variances contributed by the components.

The Sort program utilizes the internal MergeSort process in the Reduce phase to sort numeric values. A special partitioning function is used to make sure each Reduce task handle a specific range of values [13]. Both the Map function and the Reduce function just simply pass data from input to output, which eliminate the variances brought by user-defined functions. As the sample dataset is drawn from a uniform distribution, each Reduce task will get an approximately same amount of data. As a result, we can safely assume that the costs of the Map and Reduce functions keep constant for different Map and Reduce tasks.

This experiment is done with a EC2 cluster and two sets of randomly generated data in size of 64GB and 128GB, respectively. We use the MapReduce job profiler developed by Herodotou et al. [5] to record the costs of each phase. The Reduce phase is further split into three components: Shuffle, Sort, and Reduce+WriteBack.

Figure 8 shows the Map-phase cost distribution with different numbers of Reduce tasks; Figure 9 shows the corresponding Reduce-phase cost distribution. The Map-phase cost keeps almost same with changing numbers of Reduces and input data sizes, which is consistent with our analysis. The Reduce-phase cost decreases dramatically from 8 to 32 Reduces; after that, the cost reductions are minor, which indicates that simply increasing the number of Reduce tasks is not cost-effective after certain threshold. In fact, according to Table 6, the specific time cost function is $T_{sort,aws}(M, m, R) = 21.74M/m + 3.58MlogM/R + 0.05M$, which explains why the cost of Reduce phase follows this pattern. Overall, most absolute errors (standard deviations) are from the Reduce phase - the Map phase has standard deviations around 10 seconds, while the Reduce phase has standard deviations around tens to hundreds seconds.

We further look at the error distribution in the Reduce phase, in terms of the three major components: Shuffle, Sort, and Reduce+WriteBack. Figure 10 and Figure 11 show the error distributions for 64GB data and 128GB data, correspondingly, which also include the Map phase error for comprehensive analysis. We can clearly observe that the Reduce phase errors dominate the overall error distribution. In particular, Shuffle and Reduce+WriteBack (mostly WriteBack) dominate the error of Reduce phase. Both involve network I/O. In most cases, Shuffle fetches the data remotely, while in our experimental setting WriteBack will create two replicas of each data block: one for local and the other for remote. It matches our understanding that the performance of network I/O may vary significantly, and thus is difficult to predict.

As Figure 10 and 11 show, the variance of Shuffle cost drops significantly with the increasing number of Reduce tasks, while the variance of WriteBack seems not decreasing. Therefore, it is possible to improve the modeling quality by separating the WriteBack component from the whole workflow and also using 32 or more Reduce tasks for generating training examples.

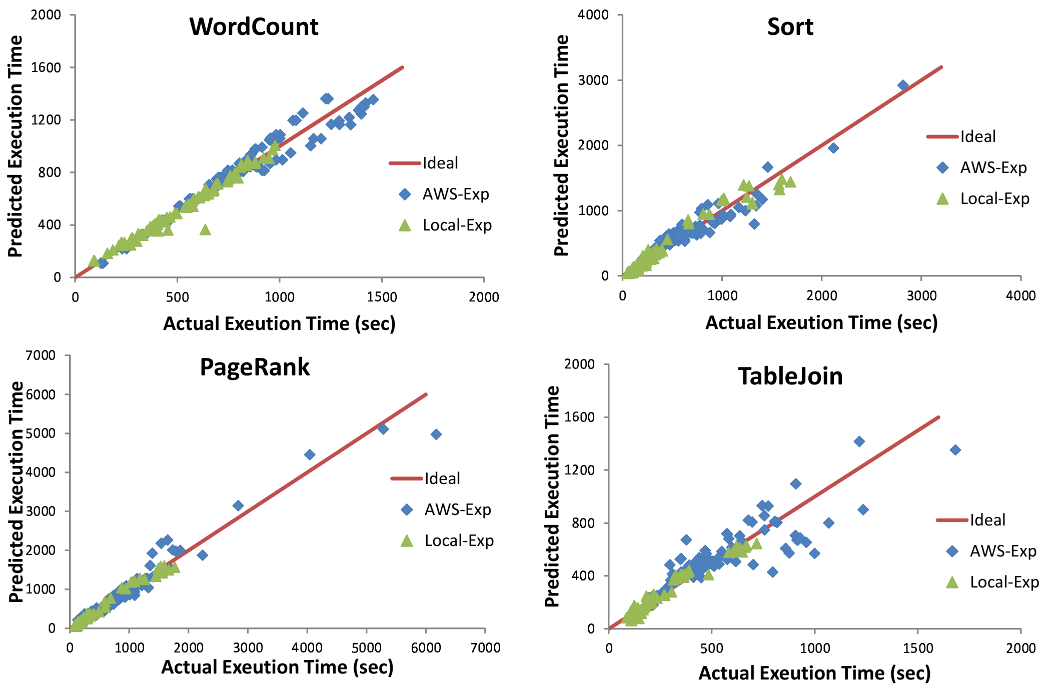
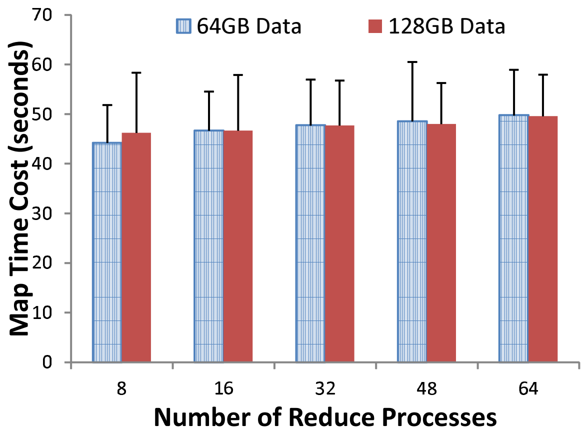


Fig. 7. A comparison on model accuracy in the local cluster and EC2 clusters.



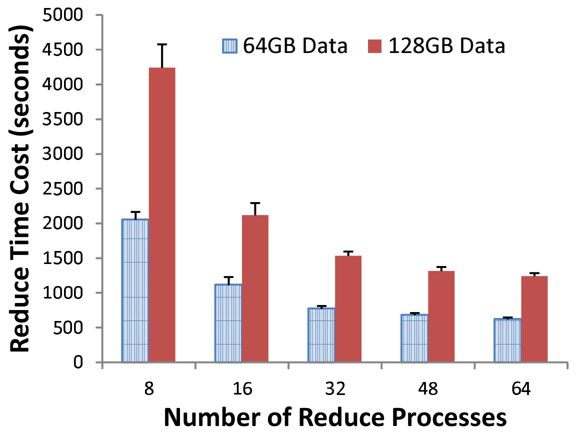


Fig. 8. Map phase cost distribution of the Sort program in the EC2 cluster.

Fig. 9. Reduce phase cost distribution of the Sort program in the EC2 cluster.

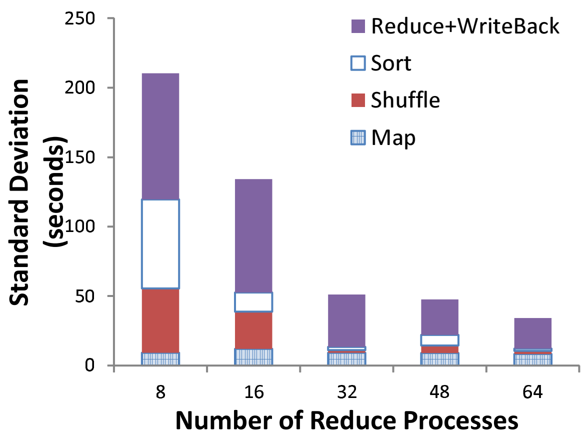


Fig. 10. The distribution of standard deviations for the Sort program with 64GB input data.

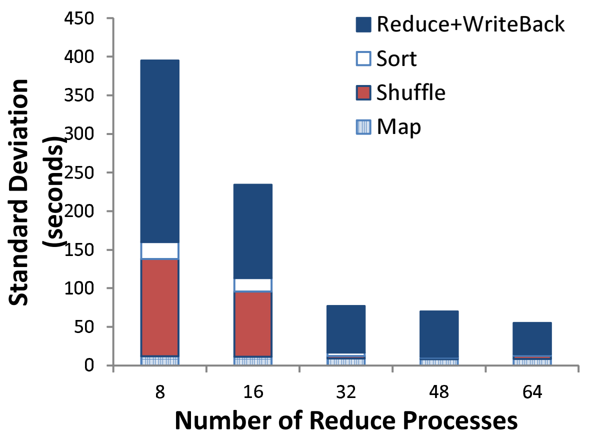


Fig. 11. Map phase cost distribution of the Sort program in the EC2 cluster.