Resource Provisioning Framework for MapReduce Jobs with Performance Goals

Abhishek Verma[1], Ludmila Cherkasova[2], and Roy H. Campbell[1]

[1] University of Illinois at Urbana-Champaign, {verma7,rhc}@illinois.edu
[2] HP Labs, Palo Alto, {lucy.cherkasova}@hp.com

Abstract. Many companies are increasingly using MapReduce for efficient large scale data processing such as personalized advertising, spam detection, and different data mining tasks. Cloud computing offers an attractive option for businesses to rent a suitable size Hadoop cluster, consume resources as a service, and pay only for resources that were utilized. One of the open questions in such environments is the amount of resources that a user should lease from the service provider. Often, a user targets specific performance goals and the application needs to complete data processing by a certain time deadline. However, currently, the task of estimating required resources to meet application performance goals is the solely user's responsibility. In this work, we introduce a novel framework and technique to address this problem and to offer a new resource sizing and provisioning service in MapReduce environments. For a MapReduce job that needs to complete within a certain time, we build the job profile by using its past executions or by executing it on a smaller data set. Then, by applying scaling rules combined with a fast and efficient capacity planning model, a set of resource provisioning options is generated. Moreover, we design a model for estimating the impact of node failures on a job completion time to evaluate worst case scenarios. We validate the accuracy of our models using a set of realistic applications. The predicted completion times of generated resource provisioning options are within 10% of the measured times in our 66-node Hadoop cluster.

1 Introduction

Private and public clouds offer a new delivery model with virtually unlimited computing and storage resources. Many companies are following the new trend of using MapReduce [1] and its open-source implementation Hadoop for large-scale, data intensive processing and for mining petabytes of unstructured information. However, setting up a dedicated Hadoop cluster requires a significant capital expenditure that can be difficult to justify. Cloud computing offers a compelling alternative and allows users to rent resources in a "pay-as-you-go" fashion. A list of supported services by Amazon (Amazon Web Services) includes MapReduce environments for rent. It is an attractive and cost-efficient option for many users because acquiring and maintaining complex, large-scale infrastructures is a difficult and expensive decision. Hence, a typical practice among MapReduce users is to develop their applications in-house using a small development testbed and test it over a small input dataset. They can lease a MapReduce cluster from the service provider and subsequently execute their MapReduce applications on large input datasets of interest. Often, the application is a part of a more elaborate business pipeline, and the MapReduce job has to produce results by a certain time deadline, i.e., it has to achieve certain performance goals and service level objectives (SLOs). Thus, a typical performance question in MapReduce environments is "*how to estimate* *the required resources (number of map and reduce slots) for a job so that it achieves certain performance goals and completes data processing by a given time?*" Currently, there is no available methodology to easily answer this question, and businesses are left on their own to struggle with the resource sizing problem: they need to perform adequate application testing, performance evaluation, capacity planning estimation, and then request appropriate amount of resources from the service provider.

In this work, we propose a novel framework to solve this problem and offer a new resource sizing and provisioning service in MapReduce environments. For a MapReduce job that needs to complete within a certain time, its job profile can be built from its past executions. Alternatively, profiling can done by executing a given application with a smaller input dataset than the original one. The power of the designed technique is that it offers a compact job profile that is comprised of performance *invariants* which are independent of the amount of resources assigned to the job (i.e., the size of the Hadoop cluster) and the size of the input dataset. The job profile accurately reflects the application performance characteristics during all phases of a given job: map, shuffle/sort, and reduce phases. For many applications, increasing input dataset while keeping the same number of reduce tasks leads to an increased amount of data shuffled and processed per reduce task. Using linear regression, we derive *scaling factors* for shuffle and reduce phases to estimate their service times as a function of input data.

We design a *MapReduce performance model* that predicts the job completion time based on the job profile, input dataset size, and allocated resources. We enhance the designed model to evaluate the performance impact of failures on job completion time. This model helps in evaluating worst case scenarios and deciding on the necessity of additional resources or program changes as a means of coping with potential failure scenarios. Finally, we propose a fast and efficient capacity planning procedure for estimating the required resources to meet a given application SLO. The output of the model is a set of plausible solutions (if such solutions exist for a given SLO) with a choice of different numbers of map and reduce slots that need to be allocated for achieving performance goals of this application.

We validate the accuracy of our approach and performance models using a set of realistic applications. First, we build the application profiles in a small staging testbed while using small input datasets for processing. Then we perform capacity planning and generate plausible resource provisioning options for achieving a given application SLO. The predicted completion times of these generated options are within 10% of the measured times in the 66-node Hadoop cluster.

This paper is organized as follows. Section 2 provides a background on MapReduce. Section 3 introduces our approach towards profiling MapReduce jobs. Section 4 presents a variety of MapReduce performance models and the SLO-based resource provisioning. The efficiency of our approach and the accuracy of designed models is evaluated in Section 5. Section 6 describes the related work. Section 7 summarizes the paper and outlines future directions.

2 MapReduce Background

This section provides an overview of the MapReduce [1] abstraction, execution, scheduling, and failure modes. In the MapReduce model, computation is expressed as two functions: map and reduce. The map function takes an input pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key k_2

are grouped together and then passed to the reduce function. The reduce function takes intermediate key k_2 with a list of values and processes them to form a new list of values.

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

MapReduce jobs are distributed and executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*.

Each map task processes a logical split of input data that generally resides on a distributed file system. The map task reads the data, applies the user-defined map function on each record, and buffers the resulting output. This data is sorted and partitioned for different reduce tasks, and written to the local disk of the machine executing the map task. The reduce stage consists of three phases: *shuffle*, *sort* and *reduce* phase. In the *shuffle phase*, the reduce tasks fetch the intermediate data files from the already completed map tasks, thus following the "pull" model. In the *sort phase*, the intermediate files from all the map tasks are sorted. An external merge sort is used in case the intermediate data does not fit in memory as follows: the intermediate data is shuffled, merged in memory, and written to disk. After all the intermediate data is shuffled, a final pass is made to merge all these sorted files. Thus, the shuffle and sort phases are interleaved. Finally, in the *reduce phase*, the sorted intermediate data is passed to the user-defined reduce function. The output from the reduce function is generally written back to the distributed file system.

Job scheduling in Hadoop is performed by a master node, which manages a number of worker nodes in the cluster. Each worker has a fixed number of *map slots* and *reduce slots*, which can run tasks. The number of map and reduce slots is statically configured (typically, one or two per core or disk). The slaves periodically send heartbeats to the master to report the number of free slots and the progress of tasks that they are currently running. Based on the availability of free slots and the scheduling policy, the master assigns map and reduce tasks to slots in the cluster.

In the real world, user code is buggy, processes crash, and machines fail. MapReduce is designed to scale to a large number of machines and to yield a graceful performance degradation in case of failures. There are three types of failures that can occur. First, a map or reduce task can fail because of buggy code or runtime exceptions. The worker node running the failed task detects task failures and notifies the master. The master reschedules the execution of the failed task, preferably on a different machine. Secondly, a worker can fail, e.g., because of OS crash, faulty hard disk, or network interface failure. The master notices a worker that has not sent any heartbeats for a specified time interval and removes it from its worker pool for scheduling new tasks. Any tasks in progress on the failed worker are rescheduled for execution. The master also reschedules all the completed map tasks on the failed worker that belong to running jobs, since the intermediate data of these maps may not be accessible to reduce tasks of these jobs. Finally, the failure of the master is the most serious failure mode. Currently, Hadoop has no mechanism for dealing with the failure of the job master. This failure is rare and can be avoided by running multiple masters and using a Paxos consensus protocol to decide the primary master.

3 Profiling MapReduce Jobs

In this section, we discuss different executions of the same MapReduce job in the Hadoop cluster as a function of job's map and reduce tasks and the allocated map and reduce slots for executing the job. Our goal is to extract a single *job profile* that uniquely captures critical performance characteristics of the job execution in different stages.

3.1 Job Execution as a Function of Allocated Resources

Let us consider two popular MapReduce applications (described below) and demonstrate the differences between their job executions and job completion times as a function of the amount of resources allocated to these jobs.

- The first application is the *Sort benchmark* [2], which involves the use of identity map/reduce function: The output of the map and reduce task is the same as its input. Thus, the entire input of map tasks is shuffled to reduce tasks and then written as output.
- The second application is *WikiTrends* application that processes Wikipedia article traffic logs that were collected (and compressed) every hour. *WikiTrends* counts the number of times each article has been visited in the given input dataset, i.e. access frequency or popularity count of each Wikipedia article over time.

First, we run the Sort benchmark with 8GB input on 64 machines each configured with a single map and a single reduce slot, i.e., with 64 map and 64 reduce slots overall. Figure 1 shows the progress of the map and reduce tasks over time (on the x-axis) vs the 64 map slots and 64 reduce slots (on the y-axis). Since we use blocksize of 128MB, we have 8GB/128MB = 64 input splits. As each split is processed by a different map task, the job consists of 64 map tasks. This job execution results in a single map and reduce *wave*. We split each reduce task into its constituent shuffle, sort and reduce phases. As seen in the figure, since the shuffle phase starts immediately after the first map task is completed, the shuffle phase overlaps with the map stage.

Next, we run the Sort benchmark with the same 8GB input dataset on the same testbed, except this time, we provide it with fewer resources: 16 map slots and 22 reduce slots. As shown in Figure 3, since the number of map tasks is greater than the number of provided map slots, the map stage proceeds in multiple rounds of slot assignment, viz. 4 waves ($\lceil 64/16 \rceil$)[3]. These waves are not synchronized with each other, and the scheduler assigns the task to the slot with the earliest finishing time. Similarly, the reduce stage proceeds in 3 waves ($\lceil 64/22 \rceil$). In Fig. 1- 4, we show the sort phase duration that is complementary to the shuffle phase.

While the executions of four map waves resemble each other, note the difference between the first reduce wave and the following two reduce waves in this job execution. As we mentioned earlier, the shuffle phase of the first reduce wave starts immediately after the first map task completes. Moreover, this first shuffle phase continues until all the map tasks are complete, and their intermediate data is copied to the active reduce tasks. Thus the first shuffle phase overlaps with the entire map stage. The sort phase of the reduce tasks occurs in parallel with the shuffle but can complete only after their shuffle phase is completed. Finally, after the sort phase is done, the reduce computation can be performed. After that, the released reduce slots become available to the next

[3] Note, that for multiple map and reduce waves, there is an extra overhead for starting map and reduce tasks. In our ongoing work, we design a set of micro-benchmarks to automatically assess these overheads in different MapReduce environments for incorporating them in the performance model.

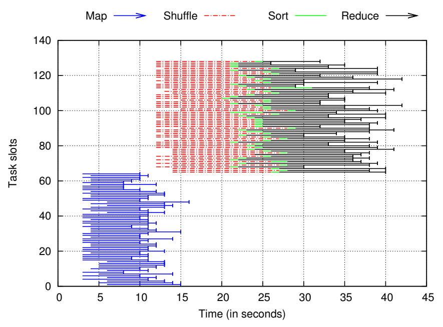


Fig. 1. Sorting with 64 map and 64 reduce slots.

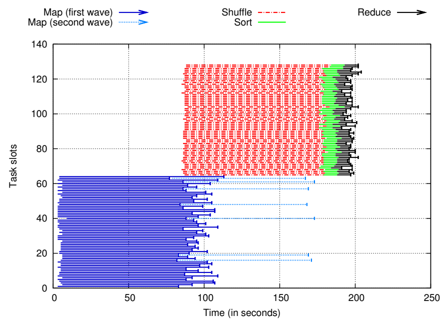


Fig. 2. WikiTrends with 64 map and 64 reduce slots.

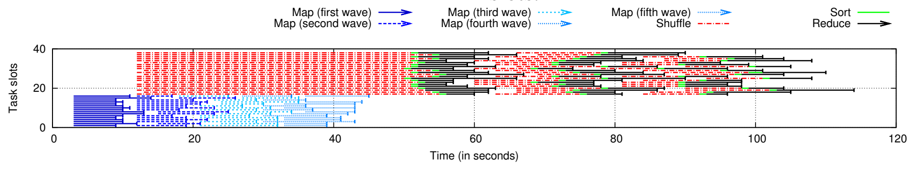


Fig. 3. Sorting with 16 map and 22 reduce slots

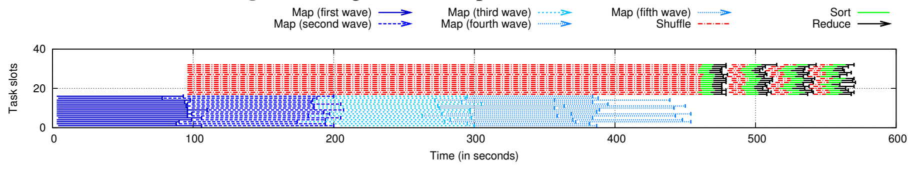


Fig. 4. WikiTrends with 16 map and 16 reduce slots.

reduce tasks. As shown in Figure 3, there is a drastic difference between the execution of the first reduce wave, but the executions of the remaining reduce waves bear a strong resemblance to each other.

Figures 2 and 4 present our second example with WikiTrends application. In this example, we process a subset of logs collected during a few days in August, 2010. There are 71 files in the set that correspond to 71 map tasks and 64 reduce tasks in this application. First, we execute WikiTrends with 64 map and 64 reduce slots. The job execution consists of two map waves ($\lceil 71/64 \rceil$) and a single reduce wave as shown in Figure 2. The second map wave processes only 7 map tasks. However, the shuffle phase of the reduce stage can be completed only when all the map tasks are done, and overlaps with both preceding map waves. Figure 4 shows the WikiTrends execution with 16 map and 16 reduce slots. The job execution has 5 map and 4 reduce waves. Again, we can see a striking difference between the first reduce wave and the remaining 3 reduce waves (which resemble each other).

As observed from Figures 1- 4, it is difficult to predict the completion time of the same job when different amount of resources are given to the job. Traditionally, a simple rule of thumb states [3], that if T is a completion time of a MapReduce job with X map and Y reduce slots then by using a smaller Hadoop cluster with $X/2$ map and $Y/2$ reduce slots the same job will be processed twice as slow, i.e., in $2 \cdot T$. While it is clear, that the job execution time is a function of allocated resources, the scaling rules are more complex, and the simple example with WikiTrends shows this. The completion time of WikiTrends in 64x64 configuration is approx. 200 sec. However, the completion time of WikiTrends in 16x16 configuration (4 times smaller cluster) is approx. 570 sec, which is far less than 4 times (naively expected) completion time increase. Apparently, more elaborate modeling and job profiling techniques are needed to capture the unique characteristics of MapReduce applications and to predict their completion time.

3.2 Job Performance Invariants as a Job Profile

Our goal is to create a compact job profile comprising of performance *invariants* that are independent of the amount of resources assigned to the job over time and that reflects all phases of a given job: map, shuffle/sort, and reduce phases. Metrics and timing of different phases, that we use below, can be obtained from the counters at the job master during the job's execution or parsed from the logs.

Map Stage: The map stage consists of a number of map tasks. To compactly characterize the distribution of the map task durations and other invariant properties, we extract the following metrics:

$$(M_{min}, M_{avg}, M_{max}, AvgSize_M^{input}, Selectivity_M), \text{where}$$

- M_{min} – the minimum map task duration. Since the shuffle phase starts when the first map task completes, we use M_{min} as an estimate for the beginning of the shuffle phase.
- M_{avg} – the average duration of map tasks to summarize the duration of a map wave.
- M_{max} – the maximum duration of the map tasks. [4] Since the shuffle phase can finish only when the entire map stage completes, i.e. all the map tasks complete, M_{max} is an estimate for a worst map wave completion time.
- $AvgSize_M^{input}$ - the average amount of input data for the map task. We use it to estimate the number of map tasks to be spawned for processing a new dataset.
- $Selectivity_M$ – the ratio of the map output size to the map input size. It is used to estimate the amount of intermediate data produced by the map stage as input to the reduce stage.

Reduce Stage: As described earlier, the reduce stage consists of the shuffle/sort and reduce phases. The shuffle phase begins only after the first map task has completed. The shuffle phase (of any reduce wave) completes when the entire map stage is complete and all the intermediate data generated by the map tasks has been shuffled to the reduce tasks and has been sorted. Since the shuffle and sort phases are interleaved, we do not consider the sort phase separately and include it in the shuffle phase. After shuffle/sort completes, the reduce phase is performed. Thus the profiles of shuffle and reduce phases are represented by the *average* and *maximum* of their tasks durations. In addition, for the reduce phase, we compute the *reduce selectivity*, denoted as $Selectivity_R$, which is defined as the ratio of the reduce output size to its input.

The shuffle phase of the first reduce wave may be significantly different from the shuffle phase that belongs to the next reduce waves (illustrated in Figure 3, 4). This happens because the shuffle phase of the first reduce wave overlaps with the entire map stage and depends on the number of map waves and their durations. Therefore, we collect two sets of measurements: (Sh_{avg}^1, Sh_{max}^1) for shuffle phase of the first reduce wave (called, *first shuffle*) and $(Sh_{avg}^{typ}, Sh_{max}^{typ})$ for shuffle phase of the other waves (called, *typical shuffle*). Since we are looking for the performance invariants that are

[4] To avoid the outliers and to improve the robustness of the measured maximum durations one can use instead the mean of a few top values.

independent of the amount of allocated resources to the job, we characterize a shuffle phase of the first reduce wave in a special way and include only the non-overlapping portions of the first shuffle in (Sh_{avg}^1 and Sh_{max}^1). Thus the job profile in the shuffle phase is characterized by two pairs of measurements: $(Sh_{avg}^1, Sh_{max}^1, Sh_{avg}^{typ}, Sh_{max}^{typ})$.

The reduce phase begins only after the shuffle phase is complete. The profile of the reduce phase is represented by the *average* and *maximum* of the reduce tasks durations and the *reduce selectivity*, denoted as $Selectivity_R$, which is defined as the ratio of the reduce output size to its input: $(R_{avg}, R_{max}, Selectivity_R)$.

4 MapReduce Performance Model

In this section, we design a MapReduce performance model that is based on *i)* the job profile and *ii)* the performance bounds of completion time of different job phases. This model can be used for predicting the job completion time as a function of the input dataset size and allocated resources.

4.1 General Theoretical Bounds

First, we establish the performance bounds for a makespan (completion time) of a given set of n tasks that is processed by k servers (or by k slots in MapReduce environments).

Let $T_1, T_2, \dots, T_n$ be the duration of n tasks of a given job. Let k be the number of slots that can each execute one task at a time. The assignment of tasks to slots is done using a simple, online, *greedy* algorithm, i.e., assign each task to the slot with the earliest finishing time.

Let $avg = (\sum_{i=1}^{n} T_i)/n$ and $max = \max_i \{T_i\}$ be the *average* and *maximum* duration of the n tasks respectively.

Makespan Theorem: The makespan of the greedy task assignment is at least $n \cdot avg/k$ and at most $(n-1) \cdot avg/k + max$.

The lower bound is trivial, as the best case is when all n tasks are equally distributed among the k slots (or the overall amount of work $n \cdot avg$ is processed as fast as possible by k slots). Thus, the overall makespan is at least $n \cdot avg/k$.

For the upper bound, let us consider the worst case scenario, i.e., the longest task $\hat{T} \in \{T_1, T_2, \dots, T_n\}$ with duration max is the last processed task. In this case, the time elapsed before the final task $\hat{T}$ is scheduled is at most the following: $(\sum_{i=1}^{n-1} T_i)/k \leq (n-1) \cdot avg/k$. Thus, the makespan of the overall assignment is at most $(n-1) \cdot avg/k + max$. [5] ■

These bounds are particularly useful when $max \ll n \cdot avg/k$, i.e., when the duration of the longest task is small as compared to the total makespan. The difference between lower and upper bounds represents the range of possible job completion times due to non-determinism and scheduling.

4.2 Bounds-based Completion Time Estimates of a MapReduce Job

Let us consider job J with a given profile either built from executing this job in a staging environment or extracted from past job executions. Let J be executed with a new dataset that is partitioned into N_M^J map tasks and N_R^J reduce tasks. Let S_M^J and S_R^J be the number of map and reduce slots allocated to job J respectively.

[5] Similar ideas were explored in the classic papers on scheduling, e.g., to characterize makespan bounds in [4].

Let M_{avg} and M_{max} be the average and maximum durations of map tasks (defined by the job J profile). Then, by Makespan Theorem, the lower and upper bounds on the duration of the entire map stage (denoted as T_M^{low} and T_M^{up} respectively) are estimated as follows:

$$T_M^{low} = N_M^J \cdot M_{avg}/S_M^J \tag{1}$$

$$T_M^{up} = (N_M^J - 1) \cdot M_{avg}/S_M^J + M_{max} \tag{2}$$

The reduce stage consists of shuffle (which includes the interleaved sort phase) and reduce phases. Similarly, Makespan Theorem can be directly applied to compute the lower and upper bounds of completion times for reduce phase (T_R^{low}, T_R^{up}) since we have measurements for average and maximum task durations in the reduce phase, the numbers of reduce tasks N_R^J and allocated reduce slots S_R^J. [6]

The subtlety lies in estimating the duration of the shuffle phase. We distinguish the non-overlapping portion of the *first shuffle* and the task durations in the *typical shuffle* (see Section 3 for definitions). The portion of the typical shuffle phase in the remaining reduce waves is computed as follows:

$$T_{Sh}^{low} = \left(\frac{N_R^J}{S_R^J} - 1\right) \cdot Sh_{avg}^{typ} \tag{3}$$

$$T_{Sh}^{up} = \left(\frac{N_R^J - 1}{S_R^J} - 1\right) \cdot Sh_{avg}^{typ} + Sh_{max}^{typ} \tag{4}$$

Finally, we can put together the formulae for the lower and upper bounds of the overall completion time of job J:

$$T_J^{low} = T_M^{low} + Sh_{avg}^1 + T_{Sh}^{low} + T_R^{low} \tag{5}$$

$$T_J^{up} = T_M^{up} + Sh_{max}^1 + T_{Sh}^{up} + T_R^{up} \tag{6}$$

Note that we can re-write Eq. 5 for T_J^{low} by replacing its parts with more detailed Eq. 1 and Eq. 3 and similar equations for sort and reduce phases as it is shown below:

$$T_J^{low} = \frac{N_M^J \cdot M_{avg}}{S_M^J} + \frac{N_R^J \cdot (Sh_{avg}^{typ} + R_{avg})}{S_R^J} + Sh_{avg}^1 - Sh_{avg}^{typ} \tag{7}$$

This presentation allows us to express the estimates for completion time in a simplified form shown below:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low}, \tag{8}$$

where $A_J^{low} = M_{avg}$, $B_J^{low} = (Sh_{avg}^{typ} + R_{avg})$, and $C_J^{low} = Sh_{avg}^1 - Sh_{avg}^{typ}$. Eq. 8 provides an explicit expression of a job completion time as a function of map and reduce slots allocated to job J for processing its map and reduce tasks, i.e., as a function of (N_M^J, N_R^J) and (S_M^J, S_R^J). The equation for T_J^{up} can be written similarly.

[6] For simplicity of explanation, we omit the normalization step of measured durations in job profile with respect to $AvgSize_M^{input}$ and $Selectivity_M$. We will discuss it next in Section 4.3.

4.3 Scaling Factors

In the previous section, we showed how to extract the job profile and use it for predicting job completion time when different amounts of resources are used. When the job is executed on a larger dataset the number of map tasks and reduce tasks may be scaled proportionally if the application structure allows it. In some cases, the number of reduce tasks is statically defined, e.g., 24 hours a day, or the number of categories (topics) in Wikipedia, etc. When the job is executed on a larger dataset while the number of reduce tasks is kept constant, the durations of the reduce tasks naturally increase as the size of the intermediate data processed by each reduce task increases. The duration of the map tasks is not impacted because this larger dataset is split into a larger number of map tasks but each map task processes a similar portion of data. The natural attempt might be to derive a single scaling factor for reduce task duration as a function of the amount of processed data, and then use it for the shuffle and reduce phase duration scaling as well. However, this might lead to inaccurate results. The reason is that the shuffle phase performs data transfer and its duration is mainly defined by the network performance. The reduce phase duration is defined by the application specific computation of the user supplied reduce function and significantly depends on the disk write performance. Thus, the duration scaling in these phases might be different. Consequently, we derive two scaling factors for shuffle and reduce phases separately, each one as a function of the processed dataset size.

Therefore in the staging environment, we perform a set of k experiments ($i = 1, 2, ..., k$) with a given MapReduce job for processing different size input datasets (while keeping the number of reduce tasks constant), and collect the job profile measurements. We derive scaling factors with linear regression in the following way. Let D_i be the amount of intermediate data for processing per reduce task, and let $Sh_{i,avg}^{typ}$ and $R_{i,avg}$ be the job profile measurements for shuffle and reduce phases respectively. Then, using linear regression, we solve the following sets of equations:

$$C_0^{Sh} + C_1^{Sh} \cdot D_i = Sh_{i,avg}^{typ}, (i = 1, 2, \cdots, k) \tag{9}$$

$$C_0^{R} + C_1^{R} \cdot D_i = R_{i,avg}, (i = 1, 2, \cdots, k) \tag{10}$$

Derived scaling factors (C_0^{Sh}, C_1^{Sh}) for shuffle phase and (C_0^R, C_1^R) for reduce phase are incorporated in the job profile. When job J processes an input dataset that leads to a different amount of intermediate data D_{new} per reduce task, its profile is updated as $Sh_{avg}^{typ} = C_0^{Sh} + C_1^{Sh} \cdot D_{new}$ and $R_{avg} = C_0^R + C_1^R \cdot D_{new}$. Similar scaling factors can be derived for maximum durations Sh_{max}^{typ} and R_{max} as well as for the first shuffle phase measurements.

4.4 Impact of Failures on the Completion Time Bounds

The performance implications of failures depend on the type of failures (discussed in Section 2). For example, *disk failures* are typical, but their performance implications for running MapReduce jobs are very mild. It is because by default, each piece of data is replicated three times, and data that resides on a failed disk can be fetched from other locations. Moreover, for each data block with a number of copies less than the default replication level, Hadoop will reconstruct the additional copies.

Worker failure is another typical type of failure, and its performance implications for a MapReduce job can be more serious. If the failure happens while the job was running, and the failed worker has either completed or in-progress job map tasks then all these map tasks need to be recomputed, since the intermediate data of these tasks might be unavailable to current or future reduce tasks. The same applies to the reduce tasks which were in progress on the failed worker: they need to be restarted on a different node.

Moreover, in order to understand the performance impact of a worker failure on job completion time, we need to consider not only *when* the failure happened, but also whether additional resources in the system can be allocated to the job to compensate for the failed worker. For example, if a worker failure happens in the very beginning of the map stage and the resources of the failed worker are immediately replenished with additional ones, then the lower and upper bounds of job completion time remain practically the same. However, if the failed worker resources are not replenished then the performance bounds are higher.

On the other hand, if a worker failure happens during the job's last wave of reduce tasks then all the completed map tasks that reside on the failed node as well as the reduce tasks that were in-progress on this node have to be re-executed, and even if the resources of the failed node are immediately replenished there are serious performance implications of this failure so late during the job execution. The latency for recomputing the map and reduce tasks of the failed node can not be hidden: this computation time is explicitly on the critical path of the job execution and is equivalent of adding entire map and reduce stage latency: $M_{max} + Sh_{max}^{typ} + R_{max}$.

Given the time of failure t_f, we try to quantify the job completion time bounds. Let us consider job J with a given profile, which is partitioned into N_M^J map tasks and N_R^J reduce tasks. Let the worker failure happen at some point of time t_f. There are two possibilities for the job J execution status at the time of failure, it is either in the map or the reduce stage. We can predict whether the failure happened during the map or reduce stage based on either using low or upper bounds of a completion time (or its average). Let us consider the computation based on the lower bound. We now describe how to approximate the number of map and reduce tasks yet to be completed in both the cases.

– *Case (1):* Let us assume that the job execution is in the map stage at time t_f, i.e., $t_f \leq T_M^{low}$. In order to determine the number of map tasks yet to be processed, we approximate the number of completed ($N_{M_{done}}^J$) and failed ($N_{M_{fail}}^J$) tasks as follows:

$$N_{M_{done}}^J \cdot M_{avg}/S_M^J = t_f \implies N_{M_{done}}^J = \lfloor t_f \cdot S_M^J / M_{avg} \rfloor$$

If there are W worker nodes in the Hadoop cluster for job J processing and one of them fails, then

$$N_{M_{fail}}^J = \lfloor N_{M_{done}}^J / W \rfloor$$

Thus, the number of map and reduce tasks yet to be processed at time t_f (denoted as N_{M,t_f}^J and N_{R,t_f}^J) are determined as follows:

$$N_{M,t_f}^J = N_M^J - N_{M_{done}}^J + N_{M_{fail}}^J \text{ and } N_{R,t_f}^J = N_R^J$$

– *Case (2):* Let us now assume that the map stage is complete, and the job execution is in the reduce stage at time t_f, $t_f \geq T_M^{low}$ and all the map tasks N_M^J are completed.

The number of completed reduce tasks $N^J_{R_{done}}$ at time t_f can be evaluated using Eq. 8:

$$B^{low}_J \cdot \frac{N^J_{R_{done}}}{S^J_R} = t_f - C^{low}_J - A^{low}_J \cdot \frac{N^J_M}{S^J_M}$$

Then the number of failed map and reduce tasks can be approximated as:

$$N^J_{M_{fail}} = \lfloor N^J_M / W \rfloor \text{ and } N^J_{R_{fail}} = \lfloor N^J_{R_{done}} / W \rfloor$$

The remaining map and reduce tasks of job J yet to be processed at time t_f are determined as follows:

$$N^J_{M,t_f} = N^J_{M_{fail}} \text{ and } N^J_{R,t_f} = N^J_R - N^J_{R_{done}} + N^J_{R_{fail}}$$

Let S^J_{M,t_f} and S^J_{R,t_f} be the number of map and reduce slots allocated to job J after the node failure. If the failed resources are not replenished, then the number of map and reduce slots is correspondingly decreased. The number of map and reduce tasks yet to be processed are N^J_{M,t_f} and N^J_{R,t_f} as shown above. Then the performance bounds on the processing time of these tasks can be computed using Eq. 5 and Eq. 6 introduced in Section 4.2. The worker failure is detected only after time δ depending on the value of the *heart beat interval*. Hence, the time bounds are also increased by the additional time delay δ.

4.5 SLO-based Resource Provisioning

When users plan the execution of their MapReduce applications, they often have some *service level objectives* (SLOs) that the job should complete within time T. In order to support the job SLOs, we need to be able to answer a complementary performance question: given a MapReduce job J with input dataset D, how many map and reduce slots need to be allocated to this job so that it finishes within T?

We observe a *monotonicity property* for MapReduce environments. Clearly, by allocating a higher number of map and reduce slots to a job, one can only decrease the job completion time. In the light of this monotonicity property, we reformulate the problem as follows. Given a MapReduce job J with input dataset D identify *minimal combinations* (S^J_M, S^J_R) of map and reduce slots that can be allocated to job J so that it finishes within time T? We consider three design choices for answering this question:

1) T is targeted as a *lower bound* of the job completion time. Typically, this leads to the least amount of resources allocated to the job for finishing within deadline T. The lower bound corresponds to an ideal computation under allocated resources and is rarely achievable in real environments.

2) T is targeted as an *upper bound* of the job completion time. Typically, this leads to a more aggressive resource allocations and might lead to a job completion time that is much smaller than T because worst case scenarios are also rare in production settings.

3) Given time T is targeted as the *average* between lower and upper bounds on job completion time. This more balanced resource allocation might provide a solution that enables the job to complete within time T.

Algorithm 1 finds the minimal combinations of map/reduce slots (S^J_M, S^J_R) for one of design choices above, e.g., when T is targeted as a *lower bound* of the job completion time. The algorithm sweeps through the entire range of map slot allocations and finds the corresponding values of reduce slots that are needed to complete the job within time T using a variation of Eq. 8 introduced in Section 4.2. The other cases when T is targeted as the upper bound and the average bound are handled similarly.

Algorithm 1 Resource Allocation Algorithm

```
Input:
Job profile of $J$
$(N_M^J, N_R^J) \leftarrow$ Number of map and reduce tasks of $J$
$(S_M, S_R) \leftarrow$ Total number of map and reduce slots in the cluster
$T \leftarrow$ Deadline by which job must be completed
Output: $P \leftarrow$ Set of plausible resouce allocations $(S_M^J, S_R^J)$
for $S_M^J \leftarrow \textsc{Min}(N_M^J, S_M)$ to $1$ do
    Solve the equation $\frac{A_J^{low}}{S_M^J} + \frac{B_J^{low}}{S_R^J} = T - C_J^{low}$ for $S_R^J$
    if $0 < S_R^J \leq S_R$ then
        $P \leftarrow P \cup (S_M^J, S_R^J)$
    else
        // Job cannot be completed within deadline $T$
        // with the allocated map slots
        Break out of the loop
    end if
end for
```

5 Evaluation

We perform our experiments on 66 HP DL145 GL3 machines. Each machine has four AMD 2.39MHz cores, 8 GB RAM and two 160GB hard disks (7200rpm SATA), but only one disk is used for Hadoop data. The machines are set up in two racks. The 1Gb network interfaces of the machines in the same rack are connected to a Gigabit Procurve 2650 switch. The racks are interconnected using a ProCurve 2900 switch. We use Hadoop 0.20.2 with two machines as job master and the DFS master. The remaining 64 machines are used as worker nodes, each configured with a single map and reduce slot (since data disk is a bottleneck). The blocksize of the file system is set to 64MB and the replication level is set to 3. We disabled speculation in all our experiments as it did not lead to any significant improvements.

In order to validate our model, we use four representative MapReduce applications:

1. **Grep:** This application processes 27GB of Wikipedia article text and counts the frequency of URLs in the entire corpus. Each map task searches for a string matching *"http://[a-zA-Z0-9./-]+"* and the reduce task counts the frequency of each URL.
2. **Sort:** The Sort application sorts 64GB of random data generated using random text writer in GridMix2[7]. It uses identity map and reduce tasks, since the framework performs the actual sorting.
3. **WikiTrends:** We use the data from Trending Topics (TT)[8]: Wikipedia article traffic logs that were collected (and compressed) every hour in the months of April to

[7] http://hadoop.apache.org/mapreduce/docs/current/gridmix.html
[8] `http://trendingtopics.org`

August 2010. Our MapReduce application counts the number of times each article has been visited according to the given input dataset, which is very similar to the job that is run periodically by TT.

4. **WordCount:** It counts the word frequencies in 27 GB of Wikipedia article text corpus. The map task tokenizes each line into words, while the reduce task counts the occurrence of each word.

5.1 Performance Invariants

In our first set of experiments, we aim to validate whether the metrics, that we chose for the inclusion in the job profile, indeed represent performance invariants across different executions of the job on the same input dataset. To this end, we execute our MapReduce jobs on the same datasets and the same Hadoop cluster but with a variable number of map and reduce slots: *i)* 64 map and 32 reduce slots, *ii)* 16 map and 16 reduce slots. The collected job profile metrics are summarized in Table 1. We observe that the average duration metrics are within 5% of each other. The maximum durations show slightly higher variance. Each experiment is performed 10 times, and again, collected metrics exhibit less than 5% variation. From these measurements, we conclude that job profile indeed accurately captures application behavior characteristics and reflect the job performance invariants.

Job	**Map slots**	**Reduce slots**	**Map Task duration (s)**			**Map Selectivity**	**1st Shuffle (s)**		**Typ. Shuffle (s)**		**Reduce (s)**		**Reduce Selectivity**
			Min	Avg	Max		Avg	Max	Avg	Max	Avg	Max	
Twitter	64	32	26	30	42	3.24	8	11	37	40	22	44	$3.2 \cdot 10^{-8}$
	16	16	26	29	43	3.24	7	10	37	41	21	41	$3.2 \cdot 10^{-8}$
Sort	64	32	2	5	16	1.00	7	13	30	50	53	75	1.00
	16	16	2	4	14	1.00	8	11	30	51	44	73	1.00
WordCount	64	32	5	34	40	1.31	8	11	24	30	11	14	0.46
	16	16	5	34	41	1.31	7	10	23	28	10	14	0.46
WikiTrends	64	32	66	99	120	9.98	13	27	115	142	26	34	0.37
	16	16	65	98	121	9.98	15	27	113	144	26	32	0.37

Table 1. Job profiles of the four MapReduce applications.

5.2 Scaling Factors

We execute WikiTrends and WordCount applications on gradually increasing datasets with a fixed number of reduce tasks for each application. Our intent is to measure the trend of the shuffle and reduce phase durations (average and maximum) and validate the linear regression approach proposed in Section 4.3. The following table gives the details of the experiments and the resulting co-efficients of linear regression, i.e., scaling factors of shuffle and reduce phase durations derived for these applications.

Parameters	**WikiTrends**	**WordCount**
Size of input dataset	4.3GB to 70GB	4.3GB to 43GB
Number of map tasks	70 to 1120	70 to 700
Number of reduce tasks	64	64
Number of map, reduce slots	64, 32	64, 32
$C^{Sh}_{0,avg}, C^{Sh}_{1,avg}$	16.08, 2.44	6.92, 0.66
$C^{Sh}_{0,max}, C^{Sh}_{1,max}$	10.75, 2.29	11.28, 0.71
$C^{R}_{0,avg}, C^{R}_{1,avg}$	11.45, 0.56	4.09, 0.22
$C^{R}_{0,max}, C^{R}_{1,max}$	7.96, 0.43	7.26, 0.24

Figure 5 shows that the trends are indeed linear for WikiTrends and WordCount. Note that the lines do not pass through the origin and hence the durations are not directly proportional to the dataset size.

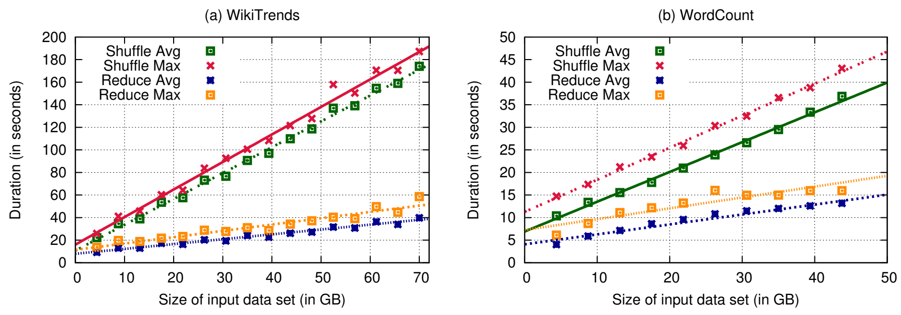


Fig. 5. Linear scaling of shuffle and reduce durations for WikiTrends and WordCount.

We observe similar results for Grep and Sort applications but do not include them in the paper due to lack of space.

5.3 Performance Bounds of Job Completion Times

In section 4, we designed performance bounds that can be used for estimating the completion time of MapReduce application with a given job profile. The expectations are that the job profile can be built using a set of job executions for processing small size input datasets, and then this job profile can be used for predicting the completion time of the same application processing a larger input dataset. Therefore, in these experiments, first, the job profiles are built using the three trials on small datasets (e.g., 4.3, 8.7 and 13.1 GB for WordCount) with different numbers of map and reduce slots. After that, by applying linear regression to the extracted job profiles from these runs, we determine the scaling factors for shuffle and reduce phases of our MapReduce jobs. The derived scaling factors are used to represent the job performance characteristics and to extrapolate the duration of the shuffle and reduce phases when the same applications are used for processing larger input datasets with parameters shown in the following table:

Parameters	**Twitter**	**Sort**	**WikiTrends**	**WordCount**
# of map tasks	370	1024	168	425
# of reduce tasks	64	64	64	64
# of map slots	64	64	64	64
# of reduce slots	16	32	8	8

Finally, by using the updated job profiles and applying the formulae described in Section 4, we predict the job completion times.

The results of these experiments are shown in Figure 6. We observe that the relative error between the predicted average T_J^{avg} and the measured job completion time is less than **10%** in all cases. However, for three applications (Sort, WikiTrends, and WordCount) the average time is below the measured one. We believe that the set of experiments in the staging environment should help to choose which bound (or weighted combination of them) should be used for more accurate estimate of the job completion time. The predicted upper bound on the job completion time T_J^{up} can be used for ensuring SLOs. The solid fill color within the bars in

Figure 6 represent the reduce stage duration, while the pattern portion reflects the duration of the map stage. For Grep, Sort, and WordCount, bounds derived from the

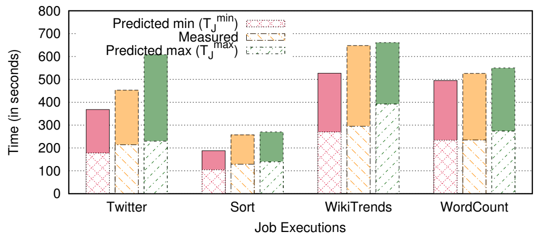


Fig. 6. Comparison of predicted and measured job completion times.

profile provide a good estimate for map and reduce stage durations. For WikiTrends, we observe a higher error in the estimation of the durations, mostly, due to the difference in processing of the unequal compressed files as inputs.

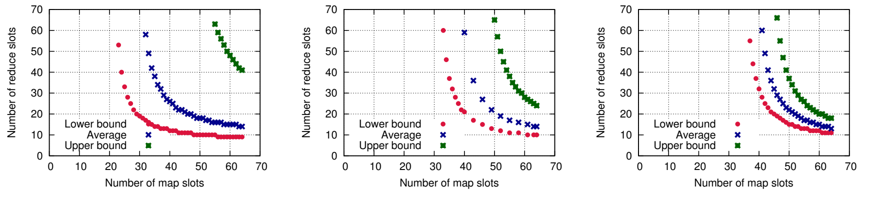
The power of the proposed approach is that it offers a compact job profile that can be derived in a small staging environment and then used for completion time prediction of the job on a large input dataset while also using different amount of resources assigned to the job.

5.4 SLO-based Resource Provisioning

In this section, we perform experiments to validate the accuracy of the SLO-based resource provisioning model introduced in Section 4.5. It operates over the following inputs *i)* a job profile built in the staging environment using smaller datasets, *ii)* the targeted amount of input data for processing, *iii)* the required job completion time. We aim to evaluate the accuracy of resource allocations recommended by the model for completing the job within a given deadline.

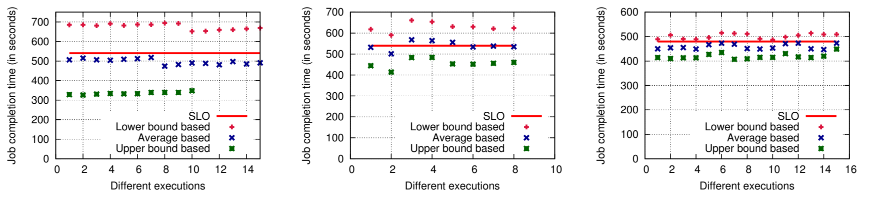
Figure 7 shows a variety of plausible solutions (the outcome of the SLO-based model) for Grep, WikiTrends and WordCount with a given deadline D= 5, 9, and 8 minutes respectively. The X and Y axes of the graph show the number of map and reduce slots respectively that need to be allocated in order to meet the job's deadline. Figure 7 presents three curves that correspond to three possible design choices for computing the required map/reduce slots as discussed in Section 4.5: when the given time T is targeted as the lower bound, upper bound, or the average of the lower and upper bounds. As expected, the recommendation based on the upper bound (worst case scenario) suggests more aggressive resource allocations with a higher number of map and reduce slots as compared to the resource allocation based on the lower bound. The difference in resource allocation is influenced by the difference between the lower and upper bounds. For example, Grep has very tight bounds which lead to more similar resource allocations based on them. For WikiTrends the difference between the lower and upper bounds of completion time estimates is wider, which leads to a larger difference in the resource allocation options.

Next, we perform a set of experiments with the applications on our 66-node Hadoop cluster. We sample each curve in Figure 7, and execute the applications with recommended allocations of map and reduce slots in our Hadoop cluster to measure the actual job completion times. Figure 8 summarizes the results of these experiments. If we base our resource computation on the lower bound of completion time, it corresponds to the "ideal" scenario. The model based on lower bounds suggests insufficient resource allocations: almost all the job executions with these allocations have missed their deadline. The closest results are obtained if we use the model that is based on the average of lower and upper bounds of completion time. However, in many cases, the measured



(a) Twitter (D = 9 mins) (b) WikiTrends (D = 9 mins) (c) WordCount (D = 8 mins)

Fig. 7. Different allocation curves based on bounds for different deadlines.



(a) Twitter (b) WikiTrends (c) WordCount

Fig. 8. Do we meet deadlines using the bounds?

completion time can exceed a given deadline (by 2-7%). If we base our computation on the upper bounds of completion time, the model over provisions resources. While all the job executions meet their deadline, the measured job completion times are lower than the target SLO, often by as much as 20%. The resource allocation choice will depend on the user goals and his requirements on how close to a given SLO the job completion time should be. The user considerations might also take into account the service provider charging schema to evaluate the resource allocation alternatives on the curves shown in Figure 7.

5.5 Prediction of Job Completion Time with Failures

In this section, we validate the model for predicting the job completion time with failures introduced in Section 4.4. For this experiment, we set the heartbeat interval to $3s$. If a heartbeat is not received in the last $20s$, the worker node is assumed to have failed. We use the WikiTrends application which consists of 720 map and 120 reduce tasks. The application is allocated 60 map and 60 reduce slots. The WikiTrends execution with given resources takes $t = 1405s$ to complete under normal circumstances. Figure 9 shows a set of two horizontal lines that correspond to lower and upper bounds of the job completion time under normal case.

Then, using the model with failures introduced in Section 4.4, we compute the lower and upper bounds for job completion time when a failure happens at time t_f (time is represented by X-axes). The model considers two different scenarios: when resources of the failed node are 1) replenished and 2) not replenished. These scenarios are reflected in Figure 9 (a) and (b) respectively. Figure 9 shows the predicted lower and upper bounds (using the model with failures) along with the measured job completion time when the worker process is killed at different points in the course of the job execution.

The shape of the lines (lower and upper bounds) for job completion time with failures is quite interesting. While the completion time with failures increases compared to the regular case, but this increase is practically constant until approximately $t = 1200s$. The map stage completes at $t = 1220(\pm 10)s$. So, the node failure during the map stage has a relatively mild impact on the overall completion time, especially when the failed

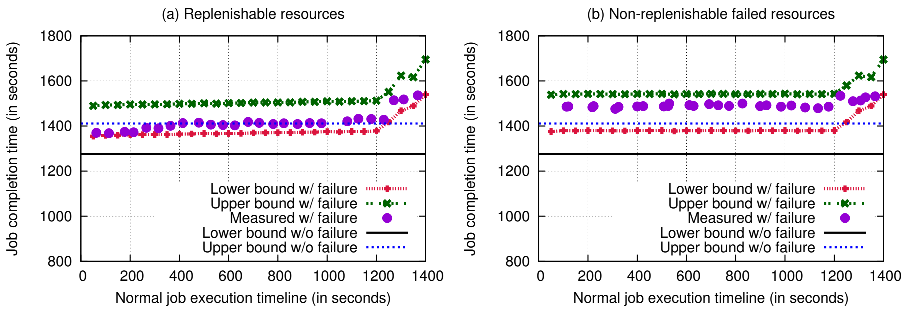


Fig. 9. Model with failures: two cases with replenishable resources and non-replenishable failed resources.

resources are replenished. However, if the failure happens in the reduce stage (especially towards the end of the job processing) then it has a more significant impact on the job completion time even if the failed node resources are replenished. Note that the measured job completion time with failures stays within predicted bounds, and hence the designed model can help the user to estimate the worst case scenario.

6 Related Work

Originally, MapReduce (and its open source implementation Hadoop) was designed for periodically running large batch workloads. With a primary goal of minimizing the job makespan the simple *FIFO* scheduler was very efficient and there was no need for special resource provisioning since the entire cluster resources could be used by the submitted job. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [5] was introduced to support more efficient cluster sharing. Capacity scheduler partitions the resources into pools and provides separate queues and priorities for each pool. By enabling partitioning of the cluster resources, the users and system administrators do need to answer an additional question: how much resources do the time-sensitive jobs require and how to translate these requirements in the capacity scheduler settings? This question is still open: there are many research efforts discussed below that aim to design a MapReduce performance model for resource provisioning and predicting the job completion time.

In order to maintain fairness between different users, the *Hadoop Fair Scheduler* (HFS) [6] allocates equal shares to each of the users running the MapReduce jobs. It also tries to maximize data locality by delaying the scheduling of the task, if no local data is available. Similar fairness and data locality goals are pursued in *Quincy* scheduler [7] proposed for the Dryad environment [8]. However, both HFS and Quincy do not provide any special support for achieving the application performance goals and the service level objectives (SLOs). Flex [9] aims to optimize some given scheduling metric and augments HFS by a special slot allocation schema. However, this approach is different from ours because it does not provide support for achieving job completion deadlines. Dynamic proportional share scheduling [10] allows users to bid for map and reduce slots by adjusting their spending over time. While this approach allows dynamically controlled resource allocation, it is driven by economic mechanisms rather than a performance model and/or application profiling.

Polo et al. [11] introduce an online job completion time estimator which can be used for adjusting the resource allocations of different jobs. However, their estimator tracks the progress of the map stage alone and has no information or control over the reduce stage. Ganapathi et al. [12] use Kernel Canonical Correlation Analysis to predict the performance of MapReduce workloads. However, they concentrate on Hive queries and do not attempt to model the actual execution of the MapReduce job. The authors discover the feature vectors through statistical correlation.

Morton et al. [13] propose *ParaTimer* for estimating the progress of parallel queries expressed as Pig scripts [14] that can translate into directed acyclic graphs (DAGs) of MapReduce jobs. In their earlier work [15], they designed *Parallax* – a progress estimator that aims to predict the completion time of a limited class of Pig queries that translate into a sequence of MapReduce jobs. In both papers, instead of a detailed profiling technique that is designed in our work, the authors rely on earlier debug runs of the same query for estimating throughput of map and reduce stages on the input data samples provided by the user. The approach is based on precomputing the expected schedule of all the tasks, and therefore identifying all the pipelines (sequences of MapReduce jobs) in the query. The approach relies on a simplistic assumption that map (reduce) tasks of the same job have the same duration. It is not clear how the authors measure the duration of reduce tasks (what phases of the reduce task are included in the measured duration), especially since the reduce task durations of the first wave and later waves are very different. Usage of the FIFO scheduler limits the approach applicability for progress estimation of multiple jobs running in the cluster with a different Hadoop scheduler, especially if the amount of resources allocated to a job varies over time or differs from the debug runs used for measurements.

Phan et al. [16] aim to build an optimal schedule for a set of MapReduce jobs with given deadlines. The authors investigate different factors that impact job performance and its completion time such as ratio of slots to core, the number of concurrent jobs, data placement, etc. MapReduce jobs with a single map and reduce waves are considered, and the scheduling problem is formulated as a constraint satisfaction problem (CSP). There are some other simplifications in MapReduce job processing where the data transfer (shuffle and sort) is considered as a separate (intermediate) phase between map and reduce tasks while in reality the shuffle phase overlaps significantly with map stage. All these assumptions and the CSP complexity issues, make it difficult to extend the proposed approach for a general case.

Cardosa at al. [17] propose a provisioning framework, called STEAMEngine, which is a family of provisioning algorithms to optimize different user or provider metrics, such as runtime, cost, throughput, or energy. STEAMEngine accumulates the database of historic observations (the same job completion times for different input dataset sizes and cluster sizes). The authors suggest to perform a few experiments for a small dataset and different cluster size combinations to enable the extrapolation technique. The profile database has separate runtimes of map and reduce phases of these job executions. While the idea of job profiling on a smaller dataset and a smaller cluster is similar to ours, the proposed profiling technique is very different. The authors only can scale the entire cluster for a job, and cannot separately scale the number of map/reduce slots. STEAMEngine relies on linear scaling which does not always work accurately as we demonstrated earlier in Section 3.

Much of the recent work also focuses on anomaly detection, stragglers and outliers control in MapReduce environments [18–22] as well as on optimization and tuning cluster parameters and testbed configuration [23, 24]. While this work is orthogonal to our research, the results are important for performance modeling in MapReduce environments. Providing more reliable, well performing, balanced environment enables reproducible results, consistent job executions and supports more accurate performance modeling and predictions.

7 Conclusion

In this work, we designed a novel framework that aims to enrich private and public clouds offering with an SLO-driven resource sizing and provisioning service in MapReduce environments. While there are quite a few companies that offer Hadoop clusters for rent, they do not provide additional performance services to answer a set of typical questions: How much resources the user application needs in order to achieve certain performance goals and complete data processing by a certain time. What is the impact of failures on the job completion time? To answer these questions, we introduced a novel profiling technique for MapReduce applications by building a compact but representative job profile in a staging environment. The approach allows executing a given application on the set of the small input datasets. Then by applying a special scaling technique and designed performance models, one can estimate the resources required for processing a targeted large dataset while meeting given SLOs. We also designed a performance model for estimating the impact of failures on MapReduce applications.

We validated the accuracy of our approach and designed performance models using a set of realistic applications in the 66-node Hadoop cluster. The accuracy of the results depends on the resource contention, especially, the network contention in the production Hadoop cluster. In our testbed (both in staging and production clusters) the network was not a bottleneck, and it led to the accurate prediction results for job completion time. Typically, service providers tend to over provision network resources to avoid undesirable side effects of network contention. At the same time, it is an interesting modeling question whether such a network contention factor can be introduced, measured, and incorporated in the proposed performance models. We also would like to automate the process of creating representative small dataset samples for job profiling from a given large input dataset. Another interesting future work is the resource provisioning of more complex applications that are defined as a composition of MapReduce jobs and meeting the SLO requirements for a given set of MapReduce jobs.

References

1. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
2. O. OMalley and A. Murthy, "Winning a 60 second dash with a yellow elephant," 2009.
3. T. White, *Hadoop:The Definitive Guide*. Page 6,Yahoo Press.
4. R. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Tech. Journal*, vol. 45, pp. 1563–1581, 1966.
5. Apache, "Capacity Scheduler Guide," 2010. [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html
6. M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of EuroSys*. ACM, 2010, pp. 265–278.

7. M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, “Quincy: fair scheduling for distributed computing clusters,” in *Proc. of the ACM SIGOPS symposium on Operating systems principles*. ACM, 2009.
8. M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “Dryad: distributed data-parallel programs from sequential building blocks,” *ACM SIGOPS OS Review*, vol. 41, no. 3, 2007.
9. J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, “FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads,” *ACM/IFIP/USENIX Intl. Middleware Conference*, 2010.
10. T. Sandholm and K. Lai, “Dynamic Proportional Share Scheduling in Hadoop,” *LNCS: Proc. of the 15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.
11. J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, “Performance-driven task co-scheduling for mapreduce environments,” in *12th IEEE/IFIP Network Operations and Management Symposium*, 2010.
12. A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, “Statistics-driven workload modeling for the cloud,” in *Proceedings of SMDB*, 2010.
13. K. Morton, M. Balazinska, and D. Grossman, “ParaTimer: a progress indicator for MapReduce DAGs,” in *Proceedings of SIGMOD*. ACM, 2010, pp. 507–518.
14. C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, “Pig latin: a not-so-foreign language for data processing,” in *Proceedings of SIGMOD*. ACM, 2008, pp. 1099–1110.
15. K. Morton, A. Friesen, M. Balazinska, and D. Grossman, “Estimating the progress of MapReduce pipelines,” in *Proceedings of ICDE*. IEEE, 2010, pp. 681–684.
16. L. Phan, Z. Zhang, B. Loo, and I. Lee, “Real-time MapReduce Scheduling,” in *Technical Report No. MS-CIS-10-32, University of Pennsylvania*, 2010.
17. M. Cardosa, P. Narang, A. Chandra, H. Pucha, and A. Singh, “Driving MapReduce Provisioning in the Cloud,” in *Technical Report TR10-023, University of Minnesota*, 2010.
18. M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, “Improving mapreduce performance in heterogeneous environments,” in *OSDI*, 2008.
19. A. Konwinski, M. Zaharia, R. Katz, and I. Stoica, “X-tracing Hadoop,” *Hadoop Summit*, 2008.
20. J. Tan, X. Pan, S. Kavulya, E. Marinelli, R. Gandhi, and P. Narasimhan, “Kahuna: Problem Diagnosis for MapReduce-based Cloud Computing Environments,” in *12th IEEE/IFIP NOMS*, 2010.
21. J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, “Visual, Log-Based Causal Tracing for Performance Debugging of MapReduce Systems,” in *ICDCS*. IEEE, 2010, pp. 795–806.
22. G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, “Reining in the Outliers in Map-Reduce Clusters using Mantri,” *OSDI*, 2010.
23. Intel, “Optimizing Hadoop* Deployments,” 2010. [Online]. Available: http://communities.intel.com/docs/DOC-4218
24. K. Kambatla, A. Pathak, and H. Pucha, “Towards optimizing hadoop provisioning in the cloud,” in *Proc. of the First Workshop on Hot Topics in Cloud Computing*, 2009.