Meeting Service Level Objectives of Pig Programs*

Zhuoyao Zhang
University of Pennsylvania
zhuoyao@seas.upenn.edu

Ludmila Cherkasova
Hewlett-Packard Labs
lucy.cherkasova@hp.com

Abhishek Verma
University of Illinois at Urbana-Champaign
verma7@illinois.edu

Boon Thau Loo
University of Pennsylvania
boonloo@cis.upenn.edu

ABSTRACT

Cloud computing offers a compelling platform to access a large amount of computing and storage resources on demand. As the technology matures, service providers have started shifting their focus to support additional user requirements such as QoS guarantees and tailored resource provisioning for achieving service performance goals. An increasing number of MapReduce applications associated with live business intelligence require completion time guarantees. We aim to solve the resource provisioning problem: given a Pig program with a completion time goal, estimate the amount of resources (a number of map and reduce slots) required for completing the program with a given (soft) deadline. We develop a simple yet elegant performance model that provides completion time estimates of a Pig program as a function of allocated resources. Then this model is used as a basis for solving the inverse resource provisioning problem for Pig programs. We evaluate our approach using a 66-node Hadoop cluster and a popular PigMix benchmark. The designed performance model accurately estimates the required amount of resources for Pig programs with completion time goals: the completion times of the Pig programs with allocated resources are within 10% of the targeted deadlines.

1. INTRODUCTION

Cloud computing has emerged as a new delivery paradigm for providing computing services with virtually unlimited computing and storage resources. The amount of enterprise data produced daily is exploding. This is partly due to a new era of automated data generation and massive event logging of digitized business processes. Many companies are following the new wave of using MapReduce [3] and its open-source implementation Hadoop to quickly process large quantities of new data to drive their core business. MapReduce offers a scalable and fault-tolerant framework for processing large data sets. However, one-input data set and simple two-stage dataflow processing schema imposed by MapReduce model is a low level and rigid. To enable programmers to specify more complex queries in an easier way, several projects, such as Pig [5], Hive [10], Scope [2], and Dryad [7], provide high-level SQL-like abstractions on top of MapReduce engines. These frameworks enable complex analytics tasks (expressed as high-level declarative abstractions) to be compiled into *directed acyclic graphs* (DAGs) of MapReduce jobs.

One of the key challenges in the cloud environment is the need to support a *performance-driven resource allocation* of cloud resources shared across multiple users running their data intensive programs. For instance, there is a technological shift towards using MapReduce and the above frameworks in support of *latency-sensitive* applications, e.g., personalized advertising, sentiment analysis, spam and fraud detection, real-time event log analysis, etc. These MapReduce applications typically require completion time guarantees and are deadline-driven. Often, they are a part of an elaborate business pipeline, and they have to produce results by a certain time deadline, i.e., to achieve certain performance goals and service level objectives (SLOs). While there have been some research efforts [12, 14, 9] towards developing performance models for MapReduce jobs, these techniques do not apply to complex queries consisting of MapReduce DAGs that characterize Pig programs.

In this paper, we study the popular Pig framework [5] and design a performance modeling environment for Pig programs as a solution to the resource provisioning problem. Specifically, given a Pig program with a completion time goal, we aim to estimate the amount of resources (a number of map and reduce slots) required for completing the Pig program with a given (*soft*) deadline. Once this estimate is known, the appropriate number of slots can be allocated to the program for the duration of its execution. As a basis for addressing this problem, we first develop a performance model that allows us to estimate the completion time of a Pig program as a function of allocated resources.

We focus on Pig, since it is quickly becoming a popular and widely-adopted system for expressing a broad variety of data analysis tasks. With Pig, the data analysts can specify complex analytics tasks without directly writing Map and Reduce functions. In June 2009, more than 40% of Hadoop production jobs at Yahoo! were Pig programs [5].

In addressing the outlined problem, the paper makes the following contributions. We propose an intuitive and efficient performance model that predicts a completion time and required resource allocation for a Pig program with performance completion time goals. Our model uses MapReduce job profiles developed automatically from previous runs

*This work was largely completed during Z. Zhang's and A. Verma's internship at HP Labs. B. T. Loo and Z. Zhang are supported in part by NSF grants (CNS-1117185, CNS-0845552 , IIS-0812270). A. Verma is supported in part by NSF grant CCF-0964471.

of the Pig program. The profiles reflect critical performance characteristics of the underlying application during its execution, and are built without requiring any modifications or instrumentation of either the application or the underlying Hadoop/Pig execution engine. All this information can be automatically obtained from the counters at the job master during the job's execution or alternatively parsed from the job logs. Our evaluation of the proposed performance model is done using the popular PigMix benchmark [1] on a 66-node Hadoop cluster. It shows that the predicted completion times are within 10% of the measured ones, and that the proposed performance model results in effective and robust resource allocation decisions.

This paper is organized as follows. Section 2 provides a background on Pig framework. Section 3 introduces a novel performance model for Pig programs with completion time goals. This model is evaluated in Section 4. Section 5 describes the related work. Section 6 presents a summary and future directions.

2. BACKGROUND: PIG PROGRAMS

There are two main components in the Pig system:

- The *language*, called Pig Latin, that combines high-level declarative style of SQL and the low-level procedural programming of MapReduce. A Pig program is similar to specifying a query execution plan: it represents a sequence of steps, where each one carries a single data transformation using a high-level data manipulation constructs, like *filter*, *group*, *join*, etc. In this way, the Pig program encodes a set of explicit dataflows.
- The *execution environment* to run Pig programs. The Pig system takes a Pig Latin program as input, compiles it into a DAG of MapReduce jobs, and coordinates their execution on a given Hadoop cluster. Pig relies on underlying Hadoop execution engine for scalability and fault-tolerance properties.

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

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

The example Pig program is compiled into a single MapReduce job. Typically, Pig programs are more complex, and can be compiled into an execution plan that consists of several stages of MapReduce jobs, some of which can run concurrently. The structure of the execution plan can be represented by a DAG of MapReduce jobs that could contain both concurrent and sequential branches. For example, Figure 1 shows a possible DAG of five MapReduce jobs $\{j_1, j_2, j_3, j_4, j_5\}$, where each node represents a MapReduce job, and the edges between the nodes represent the *data dependencies* between jobs.

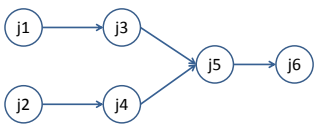

Figure 1: Example of a Pig program' execution plan represented as a DAG of MapReduce jobs.

To execute the plan, the Pig engine will first submit all the *ready* jobs (i.e., the jobs that do not have data dependency on the other jobs) to Hadoop. After Hadoop has processed these jobs, the Pig system will delete those jobs and the corresponding edges from the processing DAG, and will identify and submit the next set of ready jobs. This process continues until all the jobs are completed. In this way, the Pig engine partitions the DAG into multiple stages, each containing one or more independent MapReduce jobs that can be executed concurrently.

For example, the DAG shown in Figure 1 will be partitioned into the following four stages for processing:

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

3. PIG PERFORMANCE MODEL

We begin with a performance model for estimating the completion time of a Pig program. Then we offer the efficient solution for the inverse problem: finding the resource allocation for a Pig program with performance goals.

3.1 Performance Model of a Single MapReduce Job

As a building block for modeling Pig programs defined as DAGs of MapReduce jobs, we apply a slightly modified approach introduced in [12] for performance modeling of a single MapReduce job. The proposed MapReduce performance model [12] evaluates lower and upper bounds on the job completion time. It is based on a general model for computing performance bounds on the completion time of a given set of n tasks that are processed by k servers, (e.g., n map tasks are processed by k map slots in MapReduce environment). Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks in a given set. Let k be the number of slots that can each execute one task at a time. The assignment of tasks to slots is done using an online, *greedy* algorithm: assign each task to the slot which finished its running task the earliest. Let avg and max be the *average* and *maximum* duration of the n tasks respectively. Then the completion time of a greedy task assignment is proven to be at least:

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

and at most

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

The difference between lower and upper bounds represents the range of possible completion times due to task scheduling non-determinism (i.e., whether the maximum duration task is scheduled to run last). Note, that these provable lower and upper bounds on the completion time can be easily computed if we know the average and maximum durations of the set of tasks and the number of allocated slots. See [12] for detailed proofs on these bounds.

As motivated by the above model, in order to approximate the overall completion time of a MapReduce job J, we need to estimate the *average* and *maximum* task durations during different execution phases of the job, i.e., *map, shuffle/sort,* and *reduce* phases. Measurements such as M^J_{avg} and M^J_{max} (R^J_{avg} and R^J_{max}) of the average and maximum map (reduce) task durations for a job J can be obtained from the execution logs. By applying the outlined bounds model, we can

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

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

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

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

$$T_J^{low} = \frac{A_J^{low}}{S_M^J} + \frac{B_J^{low}}{S_R^J} + C_J^{low} \quad (3)$$

The equation for T_J^{up} can be written in a similar form (see [12] for details and exact expressions for the coefficients in these equations). Typically, the average of lower and upper bounds (T_J^{avg}) is a good approximation of the job completion time.

Once we have a technique for predicting the job completion time, it also can be used for solving the inverse problem: finding the appropriate number of map and reduce slots that could support a given job deadline D. For example, by setting the left side of Equation 3 to deadline D, we obtain Equation 4 with two variables S_M^J and S_R^J.

$$D = \frac{A_J^{low}}{S_M^J} + \frac{B_J^{low}}{S_R^J} + C_J^{low} \quad (4)$$

Note, that Equation 4 yields a hyperbola if S_M^J and S_R^J are the variables. All integral points on this hyperbola are possible allocations of map and reduce slots which result in meeting the same deadline D.

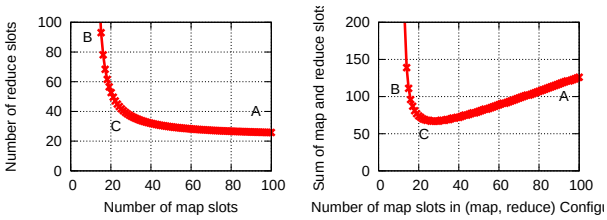

Figure 2: Lagrange curve

As shown in Figure 2 (left), the allocation could use the maximum number of map slots and very few reduce slots (shown as point A) or very few map slots and the maximum number of reduce slots (shown as point B). These different resource allocations lead to different amount of resources used (as a combined sum of allocated map and reduce slots) shown Figure 2 (right). There is a point where the sum of the map and reduce slots is minimized (shown as point C). We calculate this minima on the curve using Lagrange's multipliers [12], since we would like to conserve the number of map and reduce slots required for the minimum resource allocation per job J with a given deadline D. Note, that we can use D for finding the resource allocations from the corresponding equations for upper and lower bounds on the job completion time estimates. In Section 4, we will compare the outcome of using three alternative performance metrics for estimating a completion time of a Pig program.

3.2 Performance Model for Pig Programs

Using the model of a single MapReduce job as a building block, we consider a Pig program P that is compiled into a DAG of $|P|$ MapReduce jobs $P = \{J_1, J_2, ...J_{|P|}\}$.

Automated profiling. To automate the construction of all performance models, we build an automated profiling tool that extracts the MapReduce job profiles[1] of the Pig program from the past program executions. These job profiles represents critical performance characteristics of the underlying application during all the execution phases: map, shuffle/sort, and reduce phases.

For each MapReduce job $J_i (1 \le i \le |P|)$ that constitutes Pig program P, in addition to the number of map $(N_M^{J_i})$ and reduce $(N_R^{J_i})$ tasks, we extract metrics that reflect durations of map and reduce tasks (note that shuffle phase measurements are included in reduce task measurements) [2]:

$$(M_{avg}^{J_i}, M_{max}^{J_i}, AvgSize_M^{J_i input}, Selectivity_M^{J_i})$$

$$(R_{avg}^{J_i}, R_{max}^{J_i}, Selectivity_R^{J_i})$$

- $AvgSize_M^{J_i input}$ is the average amount of input data per map task of job J_i (we use it to estimate the number of map tasks to be spawned for processing a new dataset).

- $Selectivity_M^{J_i}$ and $Selectivity_R^{J_i}$ refer to the ratio of the map (and reduce) output size to the map input size. It is used to estimate the amount of intermediate data produced by the map (and reduce) stage of job J_i. This allows to estimate the size of the input dataset for the next job in the DAG.

Completion time bounds. Using the model outlined in Section 3.1, and the knowledge on the number of map and reduce slots $(S_M^{J_i}, S_R^{J_i})$ that were allocated for the execution of job J_i in the Pig program P, we can approximate the lower bound of completion time of each job J_i withing Pig program P, as a function of $(S_M^{J_i}, S_R^{J_i})$ $(i = 1, ..., |P|)$.

$$T_{J_i}^{low}(S_M^{J_i}, S_R^{J_i}) = \frac{A_{J_i}^{low}}{S_M^{J_i}} + \frac{B_{J_i}^{low}}{S_R^{J_i}} + C_{J_i}^{low} \quad (5)$$

Therefore, we can estimate the overall completion time of the Pig program P as a sum of completion times of all the jobs that constitute P:

$$T_P^{low} = \sum_{1 \le i \le |P|} T_{J_i}^{low}(S_M^{J_i}, S_R^{J_i}) \quad (6)$$

The computation of the estimates based on different bounds $(T_P^{up}$ and $T_P^{avg})$ are handled similarly: we use the respective models for computing T_J^{up} or T_J^{avg} for each MapReduce job $J_i (1 \le i \le |P|)$ that constitutes Pig program P.

[1] To differentiate MapReduce jobs in the same Pig program, we modified the Pig system to assign a unique name for each job as follows: *queryName-stageID-indexID*, where *stageID* represents the stage in the DAG that the job belongs to, and *indexID* represents the index of jobs within a particular stage.

[2] Unlike prior models [12], we normalize all collected measurements per record to reflect the processing cost of a single record. This normalized cost is used to approximate the duration of map and reduce tasks when the Pig program executes on a new dataset with a larger/smaller number of records. The task durations are computed by multiplying the measured per-record time by the number of input records processed by the task.

3.3 Estimating Resource Allocation

Consider a Pig program $P = \{J_1, J_2, ...J_{|P|}\}$ with a given completion time goal D. The problem is to estimate the required resource allocations (the set of map and reduce slots allocated to P during its execution) that enable the Pig program P to be completed with a (soft) deadline D.

There are a few design choices for determining the required resource allocation for a given Pig program. These choices are driven by the bound-based performance models designed in Section 3.2:

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

For example, when D is targeted as a *lower bound* of the Pig program completion time, one possible strategy is to pick a set of intermediate completion times D_i for each job J_i from the set $P = \{J_1, J_2, \ldots, J_{|P|}\}$ such that $\Sigma_{i=1}^{|P|} D_i = D$. In other words, we need to solve the following equations based on equation 5 for an appropriate pair(S_M^i, S_R^i) of map and reduce slots for each job in the DAG:

$$\begin{bmatrix} \frac{\mathsf{A}_1}{S_M^{J_1}} + \frac{\mathsf{B}_1}{S_R^{J_1}} + \mathsf{C}_1 & = & D_1 \\ \frac{\mathsf{A}_2}{S_M^{J_2}} + \frac{\mathsf{B}_2}{S_R^{J_2}} + \mathsf{C}_2 & = & D_2 \\ \vdots & & \\ \frac{\mathsf{A}_{|P|}}{S_M^{J_{|P|}}} + \frac{\mathsf{B}_{|P|}}{S_R^{J_{|P|}}} + \mathsf{C}_{|P|} & = & D_{|P|} \end{bmatrix} \quad (7)$$

where $\mathsf{A}_i = A_{J_i}^{low} \cdot N_M^{J_i}$, $\mathsf{B}_i = B_{J_i}^{low} \cdot N_R^{J_i}$ and $\mathsf{C}_i = C_{J_i}^{low}$.

However, such a solution would be difficult to implement and manage by the scheduler. When each job in a DAG requires a different allocation of map and reduce slots then it is difficult to reserve and guarantee the timely availability of the required resources.

A simpler and more elegant solution would be to determine a specially tailored resource allocation of map and reduce slots (S_M^P, S_R^P) to be allocated to the entire Pig program P, i.e., a single pair of map and reduce slots (S_M^P, S_R^P) that is allocated to each job J_i in P, $1 \le i \le |P|$ such that P would finish within a given deadline D. Specifically, equation 7 can be rewritten with the condition $S_M^{J_1} = S_M^{J_2} = \cdots = S_M^{J_{|P|}} = S_M^P$ and $S_R^{J_1} = S_R^{J_2} = \cdots = S_R^{J_{|P|}} = S_R^P$ as

$$\frac{\sum_{1 \le i \le |P|} \mathsf{A}_i}{S_M^P} + \frac{\sum_{1 \le i \le |P|} \mathsf{B}_i}{S_R^P} + \sum_{1 \le i \le |P|} \mathsf{C}_i = D \quad (8)$$

By using the Lagrange's multipliers method as described in [12], we determine the minimum amount of resources (i.e. a pair of map and reduce slots (S_M^p, S_R^p) that results in the minimum sum of the map and reduce slots) that needs to be allocated to P for completing with a given deadline D.

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

4. EVALUATION OF THE PIG MODEL

We present evaluation results to validate the accuracy of the introduced *Pig model*: (1) its ability to predict the completion time of a Pig program as a function of allocated resources, and (2) the accuracy of resource allocation decisions for meeting the specified deadlines. We describe the experimental testbed and a workload used in our case study.

4.1 Experimental Testbed and Workload

All experiments are performed on 66 HP DL145 GL3 machines. Each machine has four AMD 2.39GHz cores, 8 GB RAM and two 160GB hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We used Hadoop 0.20.2 and Pig-0.7.0 with two machines dedicated as the JobTracker and the NameNode, and remaining 64 machines as workers. Each worker is configured with 2 map and 1 reduce slots. The file system blocksize is set to 64MB. The replication level is set to 3. We disabled speculative execution since it did not lead to significant improvements in our experiments.

In our case study, we use the popular PigMix benchmark [1] that was created for testing Pig system performance. It consists of 17 Pig programs (L1-L17), which use datasets generated by the default Pigmix data generator.

In total, it generates 8 tables that create representative datasets that are commonly processed by Pig programs. PigMix benchmark operates over a variety of tables: from a small table with 500 rows to a very large table with more than 100 million records. These tables are defined by varying number of fields: one table called "widerow" contains 500 fields and others contain 6-9 fields. Moreover, two diverse distributions (zipf and uniform) are used for generating the data in these tables.

The programs in the PigMix benchmark cover a wide range of the Pig features and operators, including exploding nested data (e.g., L1), performing different join algorithms, e.g, replicate join in L2, outer join in L13, merge join in L14. These programs utilize the user defined functions (e.g., L4), or perform multi-store query (L12), or execute group-all operator (e.g., L8), etc. The compiled DAGs of the PigMix programs contain both sequential branches (e.g., L3) and concurrent branches (e.g., L11).

4.2 Completion Time Prediction

For our experiments, we create two different datasets:

- a *test dataset* that is generated by the PigMix benchmark data generator. It contains 125 million records for the largest table and has a total size around 1TB.
- an *experimental dataset* that is formed by the same layout tables as the test dataset but which are 20% larger.

We run the Pigmix benchmark on the *test dataset* and use these executions for extracting the job profiles of the corresponding Pig programs. After that, using the extracted job profiles and the designed Pig model we compute the completion time estimates of Pig programs in the PigMix benchmark for processing these two datasets (*test and experimental*) as a function of allocated resources. We validate the predicted completion times against the measured ones.

Figure 3 shows the predicted vs measured results for the PigMix benchmark that processes the *test dataset* with 128 map and 64 reduce slots. Given that the completion times of different programs in PigMix are in a broad range of 100s – 2000s, for presentation purposes and easier comparison, we **normalize** the predicted completion times with respect to the measured ones. The three bars in Figure 3 represent the predicted completion times based on the lower (T^{low}) and upper (T^{up}) bounds, and the average of them (T^{avg}). We observe that the actual completion times (shown as the straight *Measured-CT* line) of all 17 programs fall between the lower and upper bound estimates. Moreover, the predicted completion times based on the average of the upper and lower bounds are within 10% of the measured results for most cases. The worst prediction (around 20% error) is for the Pig query L11.

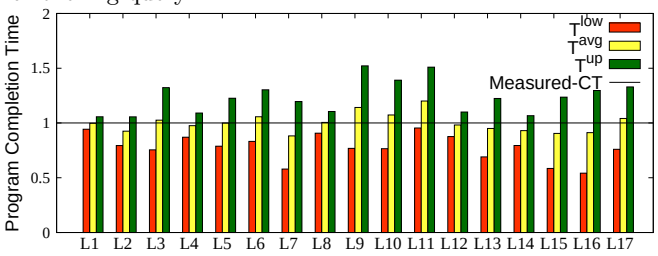

Figure 3: Predicted and measured completion time for PigMix executed with *test dataset* and 128x64 slots.

Figure 4 shows the results for the PigMix benchmark that processes the *test dataset* with 64 map and 64 reduce slots. Indeed, our model accurate computes the program completion time estimates as a function of allocated resources: the actual completion times of all 17 programs are in between the computed lower and upper bounds. The predicted completion times based on the average of the upper and lower bounds provide the best results: 10-12% of the measured results for most cases.

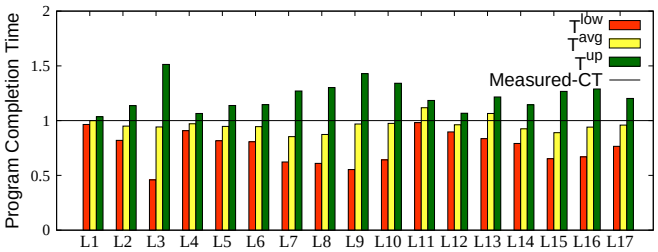

Figure 4: Predicted and measured completion time for PigMix executed with *test dataset* and 64x64 slots.

Figure 5 shows the predicted vs measured completion times for the PigMix benchmark that processes the larger, *experimental dataset* with 64 map and 64 reduce slots.

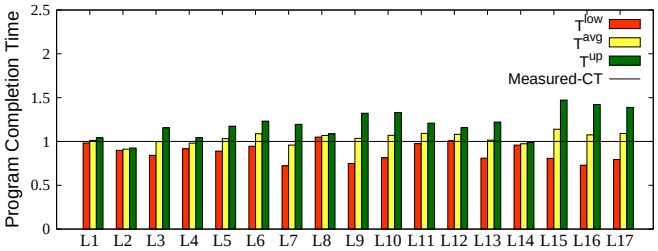

Figure 5: Predicted and measured completion time for PigMix executed with *experimental dataset* and 64x64 slots.

As shown in Figure 5, our model and computed estimates are quite accurate. The measured completion times of all programs are in between the low and upper bounds. The predicted completion times that are based on the average of the upper and lower bounds provide the best results: they are within 10% of the measured results for most cases.

In Figures 3, 4, 5 we execute PigMix benchmark three times and report the measured completion time that is averaged across 3 runs. A variance for most programs in these three runs is within 1%-2%, with the largest variance being around 6%. We have omitted the error bars in Figures 3, 4, 5 because the variance is so small.

4.3 Resource Allocation Estimation

Our second set of experiments aims to evaluate the solution of the inverse problem: the accuracy of a resource allocation for a Pig program with a completion time goal, often defined as a part of Service Level Objectives (SLOs).

In this set of experiments, let T denote the Pig program completion time when the program is processed with maximum available cluster resources (i.e., when the entire cluster is used for program processing). We set $D = 3 \cdot T$ as a completion time goal. Using the Lagrange multipliers' approach (described in Section 3.3) we compute the required resource allocation, i.e., a fraction of cluster resources, a tailored number of map and reduce slots that allow the Pig program to be completed with deadline D. As discussed in Section 3.3 we can compute a resource allocation when D is targeted as either a lower bound, or upper bound or the average of lower and upper bounds on the completion time. Figure 6 shows the measured program completion times based on these three different resource allocations. Similar to our earlier results, for presentation purposes, we **normalize** the achieved completion times with respect to deadline D.

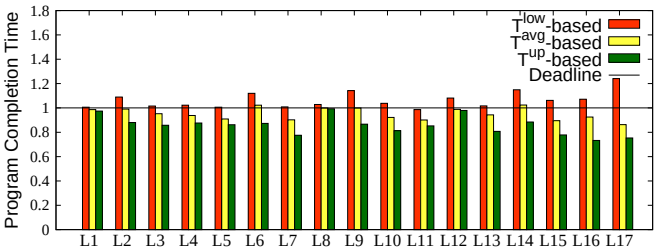

Figure 6: PigMix executed with the *test dataset*: do we meet deadlines?

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

The resource allocations based on the average between lower and upper bounds result in the closest completion time to the targeted program deadlines.

5. RELATED WORK

While performance modeling in the MapReduce framework is a new topic, there are several interesting research efforts in this direction.

Polo et al. [9] introduce an online job completion time estimator which can be used in their new Hadoop scheduler

for adjusting the resource allocations of different jobs. However, their estimator tracks the progress of the map stage alone, and use a simplistic way for predicting the job completion time, while skipping the shuffle/sort phase, and have no information or control over the reduce stage.

FLEX [14] develops a novel Hadoop scheduler by proposing a special slot allocation schema that aims to optimize some given scheduling metric. FLEX relies on the speedup function of the job (for map and reduce stages) that defines the job execution time as a function of allocated slots. However, it is not clear how to derive this function for different applications and for different sizes of input datasets. The authors do not provide a detailed MapReduce performance model for jobs with targeted job deadlines.

ARIA [12] introduces a deadline-based scheduler for Hadoop. This scheduler extracts and utilizes the job profiles from the past executions, and provides a variety of bounds-based models for predicting a job completion time as a function of allocated resources and a solution of the inverse problem. However, these models apply to a single MapReduce job.

Tian and Chen [11] aim to predict performance of a single MapReduce program from the test runs with a smaller number of nodes. They consider MapReduce processing at a fine granularity. For example, the map and reduce tasks are each partitioned in 4 functions. The authors use a *linear regression technique* to approximate the cost (duration) of each function. The problem of finding resource allocations that support given job completion goals are formulated as an optimization problem that can be solved with existing commercial solvers.

Starfish [6] applies *dynamic instrumentation* to collect a detailed run-time monitoring information about job execution at a fine granularity: data reading, map processing, spilling, merging, shuffling, sorting, reduce processing and writing. This detailed job profiling information enables the authors to analyze and predict job execution under different configuration parameters, and automatically derive an optimized configuration. One of the main challenges outlined by the authors is a design of an efficient searching strategy through the high-dimensional space of parameter values. The authors offer a workflow-aware scheduler that correlate data (block) placement with task scheduling to optimize the workflow completion time.

Ganapathi et al. [4] use Kernel Canonical Correlation Analysis to predict the performance of *Hive queries*. However, they do not attempt to model the actual execution of the MapReduce job. The authors discover the feature vectors through statistical correlation.

CoScan [13] offers a special scheduling framework that merges the execution of Pig programs with common data inputs in such a way that this data is only scanned once. Authors augment Pig programs with a set of *(deadline, reward)* options to achieve. Then they formulate the schedule as an optimization problem and offer a heuristic solution.

Morton et al. [8] propose *ParaTimer*: the progress estimator for parallel queries expressed as Pig scripts [5]. Their approach relies on a simplified assumption that map (reduce) tasks of the same job have the same duration. This work is closest to ours in pursuing the completion time estimates for Pig programs. However, the usage of the FIFO scheduler and simplifying assumptions limit 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 that are used for measurements.

6. CONCLUSION

With the increasing popularity of cloud computing, many companies are now moving towards the use of cloud infrastructures for processing large quantities of new data to drive their business applications. In this paper, we design a performance modeling framework for Pig programs as a solution to the resource provisioning problem. The proposed approach enables automated SLO-driven resource sizing and provisioning of complex workflows defined by the DAGs of MapReduce jobs.

In this work, the completion time of a Pig program is approximated by the sum of completion times of the jobs that constitute this Pig program. For a Pig program defined as a sequence of MapReduce jobs this model produces accurate results. However, such model might over-estimate the completion time of DAGs with concurrent branches (due to possible overlap of map and reduce stage executions of concurrent jobs). This may lead to the over-provisioned resource allocation. In our future work, we plan to analyze subtleties of concurrent jobs' execution and refine the designed model. Our performance models are designed for the case without node failures. We see a natural extension for incorporating different failure scenarios and estimating their impact on the application performance and achievable "degraded" SLOs. We intend to apply designed models for solving a broad set of problems related to capacity planning of MapReduce applications (defined by the DAGs of MapReduce jobs) and the analysis of various resource allocation trade-offs for supporting their SLOs.

7. REFERENCES

[1] Apache. PigMix Benchmark, http://wiki.apache.org/pig/PigMix, 2010.

[2] R. Chaiken and et al. Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. of the VLDB Endowment*, 1(2), 2008.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(1), 2008.

[4] A. Ganapathi and et al. Statistics-driven workload modeling for the cloud. In *Proc. of 5th International Workshop on Self Managing Database Systems (SMDB)*, 2010.

[5] A. Gates, O. Natkovich, S. Chopra, P. Kamath, S. Narayanam, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. of the VLDB Endowment*, 2(2), 2009.

[6] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *Proc. of CIDR'2011.*

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. *ACM SIGOPS OS Review*, 41(3), 2007.

[8] K. Morton, M. Balazinska, and D. Grossman. ParaTimer: a progress indicator for MapReduce DAGs. In *Proc. of SIGMOD.* ACM, 2010.

[9] J. Polo and et al. Performance-Driven Task Co-Scheduling for MapReduce Environments. In *In Proc. of NOMS'2010.*

[10] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive - a Warehousing Solution over a Map-Reduce Framework. *Proc. of VLDB*, 2009.

[11] F. Tian and K. Chen. Towards Optimal Resource Provisioning for Running MapReduce Programs in Public Clouds. In *Proc. of IEEE Conference on Cloud Computing (CLOUD 2011).*

[12] A. Verma, L. Cherkasova, and R. H. Campbell. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. *Proc. of ICAC'2011.*

[13] X. Wang, C. Olston, A. Sarma, and R. Burns. CoScan: Cooperative Scan Sharing in the Cloud. In *Proc. of the ACM Symposium on Cloud Computing,(SOCC'2011)*, 2011.

[14] J. Wolf and et al. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. *Proc. of the 11th ACM/IFIP/USENIX Middleware Conference*, 2010.