Performance Modeling and Optimization of Deadline-Driven Pig Programs

ZHUOYAO ZHANG, University of Pennsylvania
LUDMILA CHERKASOVA, Hewlett-Packard Labs
ABHISHEK VERMA, University of Illinois at Urbana-Champaign
BOON THAU LOO, University of Pennsylvania

Many applications associated with live business intelligence are written as complex data analysis programs defined by directed acyclic graphs of MapReduce jobs, e.g. using Pig, Hive, or Scope frameworks. An increasing number of these applications have additional requirements for completion time guarantees. In this paper, we consider the popular Pig framework that provides a high-level SQL-like abstraction on top of MapReduce engine for processing large data sets. There is a lack of performance models and analysis tools for automated performance management of such MapReduce jobs. We offer a performance modeling environment for Pig programs that automatically profiles jobs from the past runs and aims to solve the following inter-related problems: (i) estimating the completion time of a Pig program as a function of allocated resources; (ii) estimating the amount of resources (a number of map and reduce slots) required for completing a Pig program with a given (soft) deadline. First, we design a *basic* performance model that accurately predicts completion time and required resource allocation for a Pig program that is defined as a sequence of MapReduce jobs: predicted completion times are within 10% of the measured ones. Second, we optimize a Pig program execution by enforcing the *optimal schedule* of its concurrent jobs. For DAGs with concurrent jobs, this optimization helps reducing the program completion time: 10%-27% in our experiments. Moreover, it eliminates possible non-determinism of concurrent jobs' execution in the Pig program, and therefore, enables a more accurate performance model for Pig programs. Third, based on these optimizations, we propose a *refined* performance model for Pig programs with concurrent jobs. The proposed approach leads to significant resource savings (20%-60% in our experiments) compared with the original, unoptimized solution. We validate our solution using a 66-node Hadoop cluster and a diverse set of workloads: PigMix benchmark, TPC-H queries, and customized queries mining a collection of HP Labs' web proxy logs.

Categories and Subject Descriptors: C.4 [**Computer System Organization**]: Performance of Systems; C.2.6 [**Software**]: Programming Environments

General Terms: Algorithms, Design, Performance, Measurement, Management

Additional Key Words and Phrases: Hadoop, Pig, Resource Allocation, Scheduling

1. INTRODUCTION

The amount of useful enterprise data produced and collected daily is exploding. This is partly due to a new era of automated data generation and massive event logging of automated and digitized business processes; new style customer interactions that are done entirely via web; a set of novel applications used for advanced data analytics in call centers and for information management associated with data retention, government compliance rules, e-discovery and litigation issues that require to store and

This work was originated and 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.

DOI: http://dx.doi.org/10.1145/0000000.0000000

process large amount of historical data. Many companies are following the new wave of using MapReduce [Dean and Ghemawat 2008] 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. It is based on two main primitives: *map* and *reduce* functions that, in essence, perform a *group-by-aggregation* in parallel over the cluster. While initially this simplicity looks appealing to programmers, at the same time, it causes a set of limitations. One-input data set and simple two-stage dataflow processing schema imposed by MapReduce model is a low level and rigid. For analytics tasks that are based on a different dataflow, e.g., *joins* or *unions*, a special work around needs to be designed. This leads to programs with obscure semantics which are difficult to reuse. To enable programmers to specify more complex queries in an easier way, several projects, such as Pig [Gates et al. 2009], Hive [Thusoo et al. 2009], Scope [Chaiken et al. 2008], and Dryad [Isard et al. 2007], 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.

Another technological trend is the 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 [Verma et al. 2011a; Wolf et al. 2010; Polo et al. 2010] towards developing performance models for MapReduce jobs, these techniques do not apply to complex queries consisting of MapReduce DAGs. To address this limitation, our paper studies the popular Pig framework [Gates et al. 2009], and aims to design a performance modeling environment for Pig programs to offer solutions for the following problems:

- Given a Pig program, estimate its completion time as a function of allocated resources (i.e., allocated map and reduce slots);
- 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 Pig program with a given (*soft*) deadline.

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 [Gates et al. 2009]. While our paper is based on the Pig experience, we believe that the proposed models and optimizations are general and can be applied for performance modeling and resource allocations of complex analytics tasks that are expressed as an ensemble (DAG) of MapReduce jobs.

In addressing the above two problems, the paper makes the following contributions:

Basic performance model. We propose a simple and intuitive *basic* performance model that accurately predicts a completion time and required resource allocation for a Pig program that is defined as a *sequence* of MapReduce jobs. Our model uses MapReduce job profiles extracted automatically from previous runs 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 *basic* model with PigMix benchmark [Apache 2010] on a 66-node Hadoop cluster shows that the predicted completion times are within 10% of the measured ones, and the proposed model results in effective resource allocation decisions for sequential Pig programs.

Pig scheduling optimizations. When a Pig program is defined by a DAG with concurrent jobs, the proposed *basic* model is pessimistic, i.e., the predicted completion time is higher than the measured one. The reason is that unlike the execution of sequential jobs where the next job can only start after the previous one is completed, for concurrent jobs, once the previous job completes its map phase and begins its reduce phase, the next job can start its map phase execution with the released map resources in a pipelined fashion. The performance model should take this "overlap" in executions of concurrent jobs into account. Moreover, the random order of concurrent jobs' execution by Hadoop scheduler may lead to inefficient resource usage and increased processing time. Using this observation, we first, optimize a Pig program execution by enforcing the *optimal schedule* of its concurrent jobs. We evaluate optimized Pig programs and the related performance improvements using TPC-H queries and a set of customized queries mining a collection of HP Labs' web proxy logs (both sets are presented by the DAGs with concurrent jobs). Our results show 10%-27% decrease in Pig program completion times.

Refined performance model. The proposed Pig optimization has another useful outcome: it eliminates existing non-determinism in Pig program execution of concurrent jobs, and therefore, it enables better performance predictions. We develop an accurate *refined* performance model for completion time estimates and resource allocations of optimized Pig programs. We validate the accuracy of the *refined* model using a combination of TPC-H and web proxy log analysis queries. Moreover, we show that for Pig programs with concurrent jobs, this approach leads to significant resource savings (20%-60% in our experiments) compared with the original, non-optimized solution.

This paper is organized as follows. Section 2 provides a background on MapReduce and Pig frameworks. Section 3 introduces a *basic* performance model for Pig programs with completion time goals. This model is evaluated in Section 4. Section 5 proposes an optimized scheduling of concurrent jobs in Pig program and introduces a *refined* performance model. The accuracy of the new model is evaluated in Section 6. Section 7 discusses the challenges in Pig programs modeling with different size input datasets. Section 8 describes the related work. Section 9 presents a summary and future directions.

2. BACKGROUND

This section provides a basic background on the MapReduce framework [Dean and Ghemawat 2008] and its extension, the Pig system, that offers a higher-level abstraction for expressing more complex analytic tasks using SQL-style constructs.

2.1. MapReduce Jobs

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 reduce function then merges or aggregates all the values associated with the same key.

MapReduce jobs are automatically parallelized, distributed, and executed on a large cluster of commodity machines. The map and reduce stages are partitioned into map and reduce tasks respectively. 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, hence interleaving the shuffle and sort phases. 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 map and reduce tasks respectively. 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.

2.2. Pig Programs

The current Pig system consists of the following main components:

- 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 represent 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 consisting 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. 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.

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

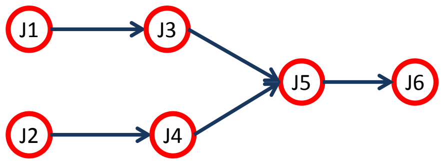


Fig. 1. Example of a Pig program' execution plan represented as a DAG of MapReduce jobs.

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: $\{j_6\}$.

In Section 4, we will show some examples based on TPC-H and web log analysis queries that are representative of such MapReduce DAGs. Note that for stages with concurrent jobs, there is no specifically defined ordering in which the jobs are going to be executed by Hadoop, For example, for a first stage in our example it could be j_1 followed by j_2, or j_2 followed by j_1. This random ordering leads to a non-determinism in the Pig program execution that might have significant performance implications as we will see in Section 5.

3. BASIC PERFORMANCE MODEL

In this section, we introduce a *basic* performance model for predicting completion time and resource allocation of a Pig program with performance goals. This model is simple, intuitive, and effective for estimating the completion times of sequential Pig programs.

3.1. Performance Model for Single MapReduce

As a building block for modeling Pig programs defined as DAGs of MapReduce jobs, we apply a slightly modified approach introduced in [Verma et al. 2011a] for performance modeling of a single MapReduce job. The proposed MapReduce performance model [Verma et al. 2011a] evaluates the 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 $T_1, T_2, \dots, T_n$ 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 the lower and upper bounds represents the range of possible completion times due to a 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 [Verma et al. 2011a] 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. These measurements are extracted from the latest past run of this job and they can be obtained from the job execution logs. By applying the outlined bounds model, we can estimate the completion times of different processing phases of the job. 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:

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

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

where M_{avg} and M_{max} are the average and maximum of the map task durations of the past run respectively. 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:

$$T_J^{low} = \frac{A_J^{low}}{S_M^J} + \frac{B_J^{low}}{S_R^J} + C_J^{low}. \tag{3}$$

The equation for T_J^{up} can be written in a similar form (see [Verma et al. 2011a] for details and exact expressions of coefficients in these equations). Typically, the average of lower and upper bounds (denoted as 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 (e.g., if D is used instead of T_J^{low} in Equation 3). When we consider S_M^J and S_R^J as variables in Equation 3 it yields a hyperbola. All integral points on this hyperbola are possible allocations of map and reduce slots which result in meeting the same deadline D. There is a point where the sum of the required map and reduce slots is minimized. We calculate this minima on the curve using Lagrange's multipliers [Verma et al. 2011a], 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 different bounds for estimating the completion time of a Pig program.

3.2. Basic Performance Model for Pig Programs

Our goal is to design a model for a Pig program that can estimate the number of map and reduce slots required for completing a Pig program with a given (soft) deadline. These estimates can be used by the SLO-based scheduler like ARIA [Verma et al. 2011a] to tailor and control resource allocations to different applications for achieving their performance goals. When such a scheduler allocates a recommended amount of map/reduce slots to the Pig program, it uses a FIFO schedule for jobs within the DAG (see Section 2.2 for how these jobs are submitted by the Pig system).

Automated profiling. Let us consider a Pig program P that is compiled into a DAG of $|P|$ MapReduce jobs $P = \{J_1, J_2, ...J_{|P|}\}$. 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 represent critical performance characteristics of the underlying application during all the execution phases: map, shuffle/sort, and reduce phases.

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

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

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

where

- $M_{avg}^{J_i}$and $M_{max}^{J_i}$ are the average and maximum map task durations of job 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);
- $M_{avg}^{J_i}$and $M_{max}^{J_i}$ are the average and maximum map task durations of job J_i;
- $Selectivity_M^{J_i}$ 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 us to estimate the size of the input dataset for the next job in the DAG.

Completion time bounds. We extract performance profiles of all the jobs in the DAG of the Pig program P from the past program executions. Then, using the model outlined in Section 3.1, we compute the lower and upper bounds of completion time of each job J_i that belongs to a Pig program P, as a function of allocated resources (S_M^P, S_R^P), where S_M^P and S_R^P be the number of map and reduce slots assigned to the Pig program P respectively.

The *basic model* for the Pig program P estimates the overall program completion time as a sum of completion times of all the jobs that constitute P:

$$T_P^{low}(S_M^P, S_R^P) = \sum_{1 \leq i \leq |P|} T_{J_i}^{low}(S_M^P, S_R^P) \tag{4}$$

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 $J_i (1 \leq i \leq |P|)$ that constitutes Pig program P.

If individual MapReduce jobs within P are assigned different number of slots, our approach is still applicable: we would need to compute the completion time estimates of individual jobs as a function of their individually assigned resources.

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 a required resource allocation (a number of map and reduce slots) that enables the Pig program P to be completed with a (soft) deadline D.

[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 [Verma et al. 2011a], we normalize all the 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. To reflect a possible skew of records per task, we collect an average and maximum number of records per task. The task durations (average and maximum) are computed by multiplying the measured per-record time by the number of input records (average and maximum) processed by the task.

First of all, there are multiple possible resource allocations that could lead to a desirable performance goal. We could have picked a set of completion times D_i for each job J_i from the set $P = \{J_1, J_2, ...J_{|P|}\}$ such that $D_1 + D_2 + ... + D_{|P|} = D$, and then determine the number of map and reduce slots required for each job J_i to finish its processing within D_i. 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 tasks 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) for the entire Pig program P (i.e., to each job J_i, $1 \leq i \leq |P|$) such that P would finish within a given deadline D.

There are a few design choices for deriving 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. The lower bound on the completion time corresponds to "ideal" computation under allocated resources and is rarely achievable in real environments. Typically, this leads to the least amount of resources that are allocated to the Pig program for finishing within deadline T.
- Determine the resource allocation when deadline D is targeted as an *upper bound* of the Pig program completion time (i.e., is a worst case scenario). 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, we need to solve the following equation for an appropriate pair(S_M^P, S_R^P) of map and reduce slots:

$$\sum_{1 \leq i \leq |P|} T_{J_i}^{low}(S_M^P, S_R^P) = D \tag{5}$$

By using the Lagrange's multipliers method as described in [Verma et al. 2011a], 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 within 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 BASIC MODEL

In this section, we present the evaluation results to validate the accuracy of the *basic model* along two dimensions: (1) ability to correctly predict the completion time of a Pig program as a function of allocated resources, and (2) make correct resource allocation decisions so that specified deadlines are met. We also describe details of our experimental testbed and three different workload sets used in the case studies.

4.1. Experimental Testbed and Workload

All experiments in this paper are performed on 66 HP DL145 GL3 machines. Each machine has four AMD 2.39GHz cores, 8 GB RAM and two 160GB 7.2K rpm SATA hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We use 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 block size 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. Incorporating speculative execution in our model is an avenue for a future work.

In case studies, we use three different workload sets: the PigMix benchmark, TPC-H queries, and customized queries for mining HPLabs web proxy logs. Below, we briefly describe each workload and our respective modifications.

PigMix. We use the well-known PigMix benchmark [Apache 2010] that was created for testing Pig system performance. It consists of 17 Pig programs (*L1-L17*) and uses datasets generated by the default Pigmix data generator. In total, 1TB of data across 8 tables are generated. The PigMix programs cover a wide range of the Pig features and operators. The data sets are generated with similar properties to Yahoo's datasets that are commonly processed using Pig. With the exception of *L11* (that contains a stage with 2 concurrent jobs), the remaining PigMix programs are DAGs of sequential jobs.

TPC-H. Our second workload is based on TPC-H [tpc 2008], a standard database benchmark for decision-support workloads. The TPC-H benchmark comes with a data generator that is used to generate the test database for queries included in the TPC-H suite. There are eight tables such as *customer, supplier, orders, lineitem, part, partsupp, nation*, and *region* that are used by TPC-H queries. The input dataset size is controlled by the scaling factor (a parameter in the data generator). The scaling factor of 1 generates 1 GB input dataset. The created data is stored in ASCII files where each file contains pipe-delimited load data for the tables defined in the TPC-H database schemas. The Pig system is designed to process a plain text and can load this data easily with its default storage function. We select 5 queries $Q3, Q5, Q8, Q10$,and $Q19$ out of 22 SQL queries from the TPC-H benchmark and express them as Pig programs. Queries $Q3$ and $Q19$ result in workflows of sequential MapReduce jobs, while the remaining queries are represented by the DAGs with concurrent MapReduce jobs[3]:

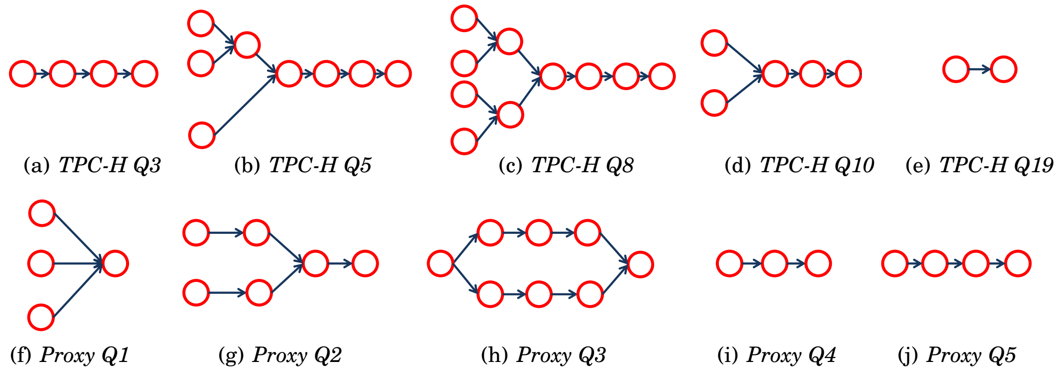


Fig. 2. DAGs of Pig programs in the TPC-H and HP Labs Proxy query sets.

- The *TPC-H Q3* retrieves the shipping priority and potential revenue of the *orders* that had not been shipped and lists them in decreasing order of revenue. It translates into 4 sequential MapReduce jobs with the DAG shown in Figure 2 (a).

[3]While more efficient logical plans may exist, our goal here is to create DAGs with concurrent jobs to stress test our model.

- The *TPC-H Q5* query lists for each *nation* in a *region* the revenue from *lineitem* transactions in which the *customer* ordering parts and the *supplier* filling them are both within that *nation*. It joins 6 tables, and its dataflow results in 3 concurrent MapReduce jobs. The DAG of the program is shown in Figure 2 (b).
- The *TPC-H Q8* query determines how the market share of a given *nation* within a given *region* has changed over two years for a given *part* type. It joins 8 tables, and its dataflow results in two stages with 4 and 2 concurrent MapReduce jobs respectively. The DAG of the program is shown in Figure 2 (c).
- The *TPC-H Q10* query identifies the *customers*, who have returned *parts* that effect the lost revenue for a given quarter. It joins 4 tables, and its dataflow results in 2 concurrent MapReduce jobs with the DAG of the program shown in Figure 2 (d).
- The *TPC-H Q19* query reports gross discounted revenue for all *orders* for three different types of *parts* that were shipped by air or delivered in person. It joins 2 tables, and its dataflow results in 2 sequential MapReduce jobs with the DAG of the program shown in Figure 2 (e).

HP Labs' Web Proxy Query Set. Our third workload consists of a set of Pig programs for analyzing HP Labs' web proxy logs. The dataset contains 6 months access logs to web proxy gateway at HP Labs during 2011-2012 years. The total dataset size (12 months) is about 36 GB. There are 438 million records in these logs, The proxy log data contains one record per each web access. The fields include information such as *date, time, time-taken, c-ip, cs-host*, etc. The log files are stored as plain text and the fields are separated with spaces. Our main intent is to evaluate our models using realistic Pig queries executed on real-world data. We aim to create a diverse set of Pig programs with dataflows with sequential and concurrent jobs.

- The *Proxy Q1* program investigates the dynamics in access frequencies to different websites per month and compares them across the 6 months. The Pig program results in 6 concurrent MapReduce jobs with the DAG shown in Figure 2 (f).
- The *Proxy Q2* program aims to discover the co-relationship between two websites from different sets (tables) of popular websites: the first set is created to represent the top 500 popular websites accessed by web users within the HPLabs. The second set contains the top 100 popular websites in US according to Alexa's statistics [4]. The DAG of the Pig program is shown in Figure 2 (g).
- The *Proxy Q3* program presents the intersect of 100 most popular websites (i.e., websites with highest access frequencies) accessed during the work and after work hours. The DAG of the program is shown in Figure 2 (h).
- The *proxy Q4* programs computes the intersection between the top 500 popular websites accessed by the HPLabs users and the top 100 popular web-sites in US. The program is translated into 3 sequential jobs with the DAG shown in Figure 2 (i).
- The *proxy Q5* program compares the average daily access frequency for each website during years 2011 and 2012 respectively. The program is translated into 4 sequential jobs. The DAG of the program is shown in Figure 2 (j).

To perform the validation experiments, we create two different datasets for each of the three workload sets described above:

(1) A **test dataset** is used for extracting the job profiles of the corresponding Pig programs.
   — PigMix: the *test dataset* is generated by the default data generator. It contains 125 million records for the largest table and has a total size around 1 TB.

[4]http://www.alexa.com/topsites

- — TPC-H: the *test dataset* is generated with scaling factor 9 using the standard data generator. The dataset size is around 9 GB.
- — HP Labs' Web proxy query set: we use the logs from February, March and April as the *test dataset* with the total input size around 18 GB.

(2) An **experimental dataset** is used to validate our performance models using the profiles extracted from the Pig programs that were executed with the *test dataset*. Both the *test* and *experimental* datasets are formed by tables with the same layout but with different input sizes.

- — PigMix: the input size of the *experimental dataset* is 20% larger than the *test dataset* (with 150 million records for the largest table).
- — TPC-H: the *experimental dataset* is around 15 GB (scaling factor 15 using the standard data generator).
- — HP Labs' Web proxy query set: we use the logs from May, June and July as the *experimental dataset*, the total input size is around 18 GB.

4.2. Completion Time Prediction

This section aims to evaluate the accuracy of the proposed *basic* performance model, i.e., whether the proposed bound-based model is capable of predicting the completion time of Pig programs as a function of allocated resources.

First, we run the three workloads on the test datasets, and the job profiles of the corresponding Pig programs are built from these executions. By using the extracted job profiles and the designed Pig model, we compute the completion time estimates of Pig programs for processing these two datasets (test and experimental) as a function of allocated resources. Then we validate the predicted completion times against the measured ones. We execute each Pig program three times and report the measured completion time averaged across 3 runs. A standard deviation for most programs in these runs is 1%-2%, with the largest value being around 6%. Since the deviation is so small, we omit the error bars in the figures.

Figure 3 shows the results for the PigMix benchmark that processes the *test dataset* when each program in the set is processed 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*. The measured completion time of *L11* is very close to the lower bound. Note, that the *L11* program is the only program in PigMix that is defined by a DAG with concurrent jobs.

Figure 4 shows the results for the PigMix benchmark that processes the *experimental dataset* when each program in the set is processed with **64** map and **64** reduce slots. Indeed, our model accurately predicts the program completion time 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.

To get a better understanding whether the *basic* model can accurately predict completion times of Pig programs with concurrent jobs, we perform similar experiments for TPC-H and Proxy queries.

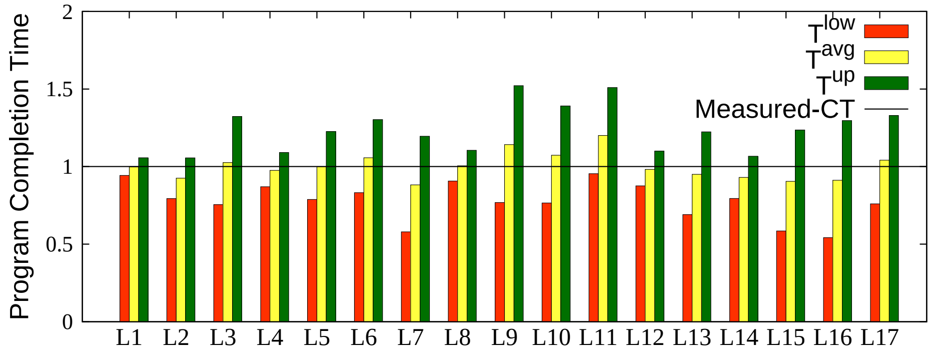


Fig. 3. Predicted and measured completion time for PigMix executed with *test dataset* and **128x64** slots.

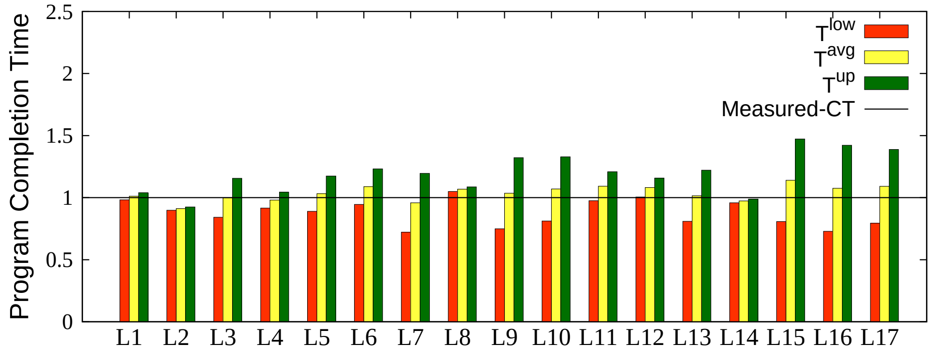


Fig. 4. Predicted and measured completion time for PigMix executed with *experimental dataset* and **64x64** slots.

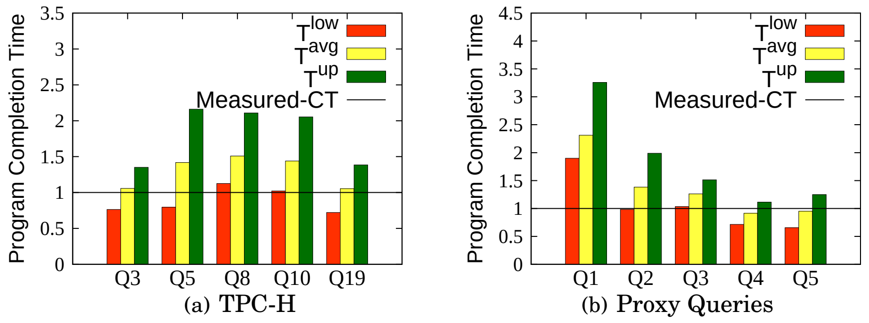


Fig. 5. Completion time estimates for TPC-H and Proxy queries executed with *experimental dataset* and **128x64** slots.

Figure 5 shows predicted completion times for TPC-H and Proxy queries processed on experimental dataset with **128** map and **64** reduce slots. For queries represented by sequential DAGs, such as *TPC-H Q3,Q19*, and *Proxy Q4,Q5*, the *basic* model estimates the program completion time accurately and measured results are close to the average bound (within 10%). For queries with concurrent jobs, such as *TPC-H Q5, Q10*, and *Proxy Q2, Q3*, the measured completion times are closer to the lower bound, while for queries *TPC-H Q8* and *Proxy Q1*, the measured program completion time is even smaller than the lower bound on the predicted completion time.

Overall, our results show that the proposed *basic* model provides good estimates for sequential DAGs, but it over-estimates completion times of programs with concurrent jobs, and leads to inaccurate prediction results for these types of Pig programs. In the upcoming Section 5, we will analyze the specifics of concurrent job executions to improve our modeling results.

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 for 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 earlier results, for presentation purposes, we **normalize** the achieved completion times with respect to deadline D.

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 of lower and upper bounds result in the closest completion time to the targeted program deadlines.

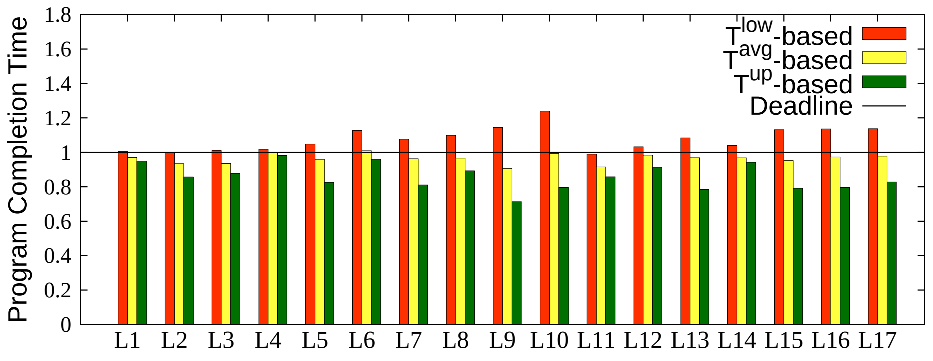
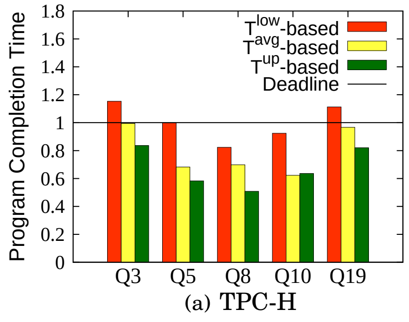
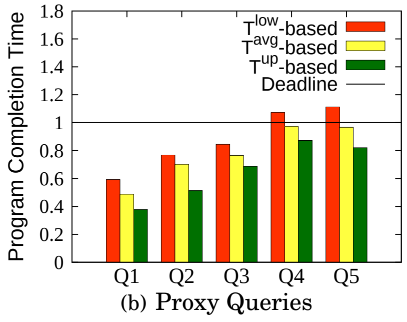


Fig. 6. PigMix executed with the *experimental dataset*: do we meet deadlines?

Figure 7 presents similar results for two other workloads: TPC-H and Proxy queries. For queries represented by sequential DAGs, such as *TPC-H Q3,Q19*, and *Proxy Q4,Q5*, the resource allocation based on the average bound results is the closest completion time to the targeted program deadlines. However, for programs with concurrent jobs we observe that the *basic* model is inaccurate. There is a significant resource over-provisioning: the considered Pig programs finish much earlier (up to 50% earlier) than the targeted deadlines.



(a) TPC-H



(b) Proxy Queries

Fig. 7. TPC-H/Proxy queries with the *experimental dataset*: do we meet deadlines?

In summary, while the *basic* model produces good results for Pig programs with *sequential* DAGs, it over-estimates the completion time of DAGs with *concurrent* jobs, and it leads to over-provisioned resource allocations for such programs.

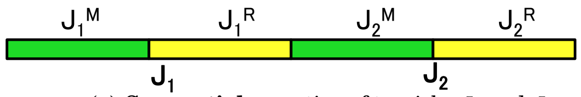
In the next Section 5, we discuss essential differences in performance modeling of concurrent and sequential jobs and offer a *refined* model.

5. REFINED PERFORMANCE MODEL FOR PIG PROGRAMS

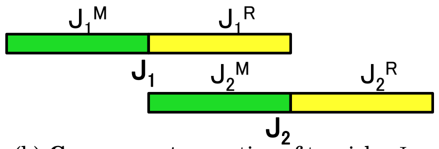
In this section, we analyze the subtleties in execution of concurrent MapReduce jobs and demonstrate that the execution order of concurrent jobs might have a significant impact on the program completion time. We optimize a Pig program execution by enforcing the *optimal schedule* of its concurrent jobs, and then introduce an accurate *refined* performance model for Pig programs and their resource allocations.

5.1. Modeling Concurrent Jobs' Executions

Let us consider two concurrent MapReduce jobs J_1 and J_2. There are no data dependencies among the concurrent jobs. Therefore, unlike the execution of sequential jobs where the next job can only start after the previous one is entirely finished (shown in Figure 8 (a)), for concurrent jobs, once the previous job completes its map phase and begins reduce phase processing, the next job can start its map phase execution with the released map resources in a pipelined fashion (shown in Figure 8 (b)). The performance model should take this "overlap" in executions of concurrent jobs into account. Moreover, this opportunity enables a better resource usage: while one job performs its map tasks processing - the other job may concurrently process their reduce tasks, etc.



(a) **Sequential** execution of two jobs J_1 and J_2.



(b) **Concurrent** execution of two jobs J_1 and J_2.

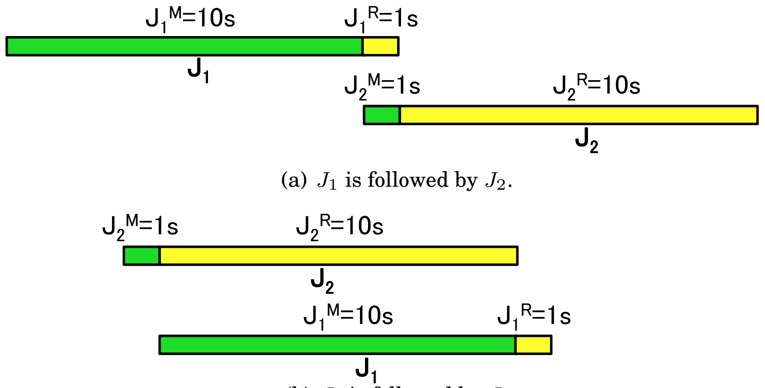
Fig. 8. Difference in executions: (a) two **sequential** MapReduce jobs; (b) two **concurrent** MapReduce jobs.

Note, that the *basic* performance model proposed in Section 3 approximates the completion time of a Pig program $P = \{J_1, J_2\}$ as a sum of completion times of J_1 and J_2 (see eq. 4) *independent on whether jobs J_1 and J_2 are sequential or concurrent*. While such an approach results in straightforward computations, at the same time, if we do not consider possible overlap in execution of map and reduce stages of concurrent jobs then the computed estimates are pessimistic and over-estimate their completion time.

We find one more interesting observation about concurrent jobs' execution of the Pig program. The original Hadoop implementation executes concurrent MapReduce jobs from the same Pig program in a random order. Some ordering may lead to a significantly less efficient resource usage and an increased processing time. Consider the following example with two concurrent MapReduce jobs:

— Job J_1: a map stage duration $J_1^M = 10s$ and the reduce stage duration $J_1^R = 1s$.
— Job J_2: a map stage duration $J_2^M = 1s$ and the reduce stage duration $J_2^R = 10s$.

There are two possible executions of J_1 and J_2 shown in Figure 9: J_1 is followed by J_2 (see Figure 9 a) and J_2 is followed by J_1(see Figure 9 b) .



(a) J_1 is followed by J_2.

(b) J_2 is followed by J_1.

Fig. 9. Impact of concurrent job scheduling on their completion time.

— J_1 is followed by J_2. The reduce stage of J_1 overlaps with the map stage of J_2 leading to overlap of only $1s$. The total completion time of processing two jobs is $10s + 1s + 10s = 21s$.
— J_2 is followed by J_1. The reduce stage of J_2 overlaps with the map stage of J_1 leading to a much better pipelined execution and a larger overlap of 10s. The total makespan is $1s + 10s + 1s = 12s$.

There could be a significant difference in the job completion time (75% in the considered example) depending on the execution order of the jobs.

We aim to optimize a Pig program execution by enforcing the optimal schedule of its concurrent jobs. We apply the classic Johnson algorithm for building the optimal two-stage jobs' schedule [Johnson. 1954]. The optimal execution of concurrent jobs leads to the improved completion time. Moreover, this optimization eliminates possible non-determinism in Pig program execution, and enables more accurate completion time predictions for Pig programs.

In the next two sections, we describe a *refined* performance model for Pig programs.

5.2. Completion Time Estimates for Pig Programs with Concurrent Jobs

Let us consider a Pig program P that is compiled into a DAG of MapReduce jobs. This DAG represents the Pig program execution plan. The Pig engine partitions the DAG into multiple stages, where each stage contains one or more independent MapReduce jobs which can be executed concurrently. For a plan execution, the Pig system first submits all the jobs from the first stage. Once they are completed, it submits jobs from the second stage, etc. This process continues until all the jobs are completed.

Note that due to the data dependencies in a Pig execution plan, the *next stage* cannot start until the *previous stage* finishes. Thus, the completion of a Pig program P which contains S *stages* can be estimated as follows:

$$T_P = \sum_{1 \le i \le S} T_{S_i} \tag{6}$$

where T_{S_i} represents the completion time of stage i.

For a stage that consists of a single job J, the stage completion time is defined by the job J's completion time. For a stage that contains concurrent jobs, the stage completion time depends on the jobs' execution order.

Suppose there are $|S_i|$ jobs within a particular stage S_i and the jobs are executed according to the order $\{J_1, J_2, ...J_{|S_i|}\}$. Note, that given a number of map/reduce slots (S_M^P, S_R^P), we can estimate the durations of map and reduce phases of MapReduce jobs $J_i (1 \le i \le |S_i|)$. These map and reduce phase durations are required by the Johnson's algorithm [Johnson. 1954] to determine the optimal schedule of jobs in the set $\{J_1, J_2, ...J_{|S_i|}\}$.

Let us assume, that for each stage with concurrent jobs, we have already determined the optimal job schedule that minimizes the completion time of the stage. Now, we introduce the *refined* performance model for predicting the Pig program P completion time T_P as a function of allocated resources (S_M^P, S_R^P). We use the following notations:

$timeStart_{J_i}^M$	the start time of job J_i's map phase
$timeEnd_{J_i}^M$	the end time of job J_i's map phase
$timeStart_{J_i}^R$	the start time of job J_i's reduce phase
$timeEnd_{J_i}^R$	the end time of job J_i's reduce phase

Then the stage completion time can be estimated as

$$T_{S_i} = timeEnd_{J_{|S_i|}}^R - timeStart_{J_1}^M \tag{7}$$

We now explain how to estimate the start/end time of each job's map/reduce phase[5].

Let $T_{J_i}^M$ and $T_{J_i}^R$ denote the completion times of map and reduce phases of job J_i respectively. We can compute phase duration as a function of allocated map/reduce slots (S_M^P, S_R^P), e.g., using a lower or upper bound estimates as described in Section 2. For the rest of the paper, we use the completion time estimates based on the average of the lower and upper bounds. Then

$$timeEnd_{J_i}^M = timeStart_{J_i}^M + T_{J_i}^M \tag{8}$$

$$timeEnd_{J_i}^R = timeStart_{J_i}^R + T_{J_i}^R \tag{9}$$

Figure 10 shows an example of three concurrent jobs execution in the order J_1, J_2, J_3.

[5]These computations present the main, typical case when the number of allocated slots is smaller than the number of tasks that jobs need to process, and therefore, the execution of concurrent jobs is pipelined. There are some corner cases with small concurrent jobs when there are enough resources for processing them at the same time. In this case, the designed model over-estimates the stage completion time. We have an additional set of equations that describes these corner cases in a more accurate way. We omit them here for a presentation simplicity.

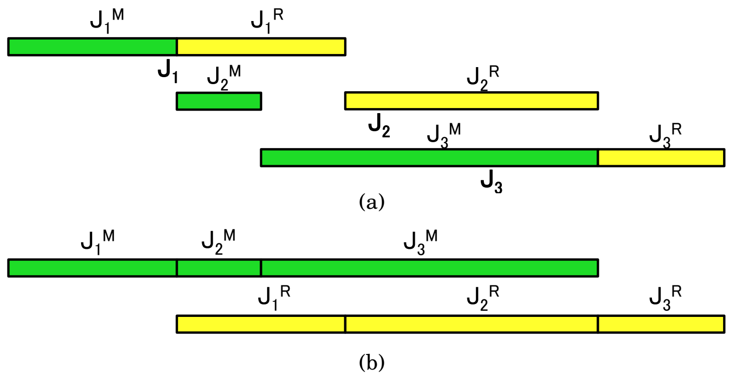


Fig. 10. Execution of Concurrent Jobs

Note, that Figure 10 (a) can be rearranged to show the execution of jobs' map/reduce stages separately (over the map/reduce slots) as shown in Figure 10 (b). It is easy to see that since all the concurrent jobs are independent, the map phase of the next job can start immediately ones the previous job's map stage is finished, i.e.,

$$timeStart_{J_i}^M = timeEnd_{J_{i-1}}^M = timeStart_{J_{i-1}}^M + T_{J_{i-1}}^M \tag{10}$$

The start time $timeStart_{J_i}^R$ of the reduce stage of the concurrent job J_i should satisfy the following two conditions:

(1) $timeStart_{J_i}^R \geq timeEnd_{J_i}^M$
(2) $timeStart_{J_i}^R \geq timeEnd_{J_{i-1}}^R$

Therefore, we have the following equation:

$$\begin{aligned} timeStart_{J_i}^R &= max\{timeEnd_{J_i}^M, timeEnd_{J_{i-1}}^R\} = \\ &= max\{timeStart_{J_i}^M + T_{J_i}^M, timeStart_{J_{i-1}}^R + T_{J_{i-1}}^R\} \end{aligned} \tag{11}$$

Then the completion time of the entire Pig program P is defined as the *sum of its stages* using eq. (6). This approach offers a *refined* performance model for Pig programs.

5.3. Resource Allocation Estimates for Optimized Pig Programs

Let us consider a Pig program P with a given deadline D. The optimized execution of concurrent jobs in P may significantly improve the program completion time. Therefore, P may need to be assigned a smaller amount of resources for meeting a given deadline D compared to its non-optimized execution. As shown in Section 4, the earlier proposed *basic* model over-estimates the completion time for Pig programs with concurrent jobs. However, the unique benefit of this model is that it allows to express the completion time D of a Pig program via a special form equation shown below:

$$D = \frac{A^P}{S_M^P} + \frac{A^P}{S_R^P} + C^P \tag{12}$$

where S_M^P and S_R^P denote the number of map and reduce slots assigned to P. Eq. (12) can be used for solving the inverse problem of finding the resource allocation (S_M^P, S_R^P) such that Pig program P completes within time D. This equation yields a hyperbola if S_M^J and S_R^J are considered as variables. We can directly calculate the minima on this curve using Lagrange's multipliers (see Section 3.1).

The *refined* model introduced in the previous Section 5.2 for predicting the completion time of an optimized Pig program is more complex. It requires computing a function *max* for stages with concurrent jobs, and therefore, it cannot be expressed as a single equation for solving the inverse problem. However, we can use the "over-sized" resource allocation defined by the *basic model* and eq. (12) as an initial point for determining the solution required by the optimized Pig program P. The hyperbola with all the possible solutions according to the *basic* model is shown in Figure 11 as the red curve, and $A(M, R)$ represents the point with a minimal number of map and reduce slots (i.e., the pair (M, R) results in the minimal sum of map and reduce slots).

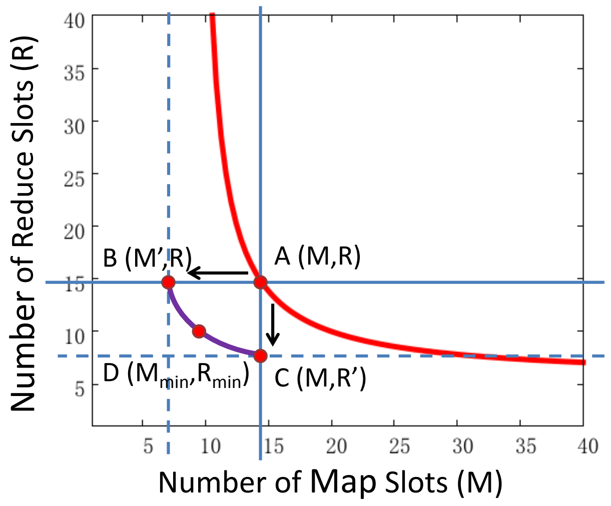


Fig. 11. Resource allocation estimates for an optimized Pig program.

Algorithm 1 described below shows the computation for determining the minimal resource allocation pair (M_{min}, R_{min}) for an optimized Pig program P with deadline D. This computation is illustrated by Figure 11.

First, we find the minimal number of map slots M' (i.e., the pair (M', R)) such that deadline D can still be met by the optimized Pig program with the enforced optimal execution of its concurrent jobs. We do it by fixing the number of reduce slots to R, and then step-by-step reducing the allocation of map slots. Specifically, Algorithm 1 sets the resource allocation to $(M - 1, R)$ and checks whether program P can still be completed within time D (we use T_P^{avg} for completion time estimates). If the answer is positive, then it tries $(M - 2, R)$ as the next allocation. This process continues until point $B(M', R)$ (see Figure 11) is found such that the number M' of map slots cannot be further reduced for meeting a given deadline D (lines 1-4 of Algorithm 1).

At the second step, we apply the same process for finding the minimal number of reduce slots R' (i.e., the pair (M, R')) such that the deadline D can still be met by the optimized Pig program P (lines 5-7 of Algorithm 1).

At the third step, we determine the intermediate values on the curve between (M', R) and (M, R') such that deadline D is met by the optimized Pig program P. Starting from point (M', R), we are trying to find the allocation of map slots from M' to M, such that the minimal number of reduce slots $\hat{R}$ should be assigned to P for meeting its deadline (lines 10-12 of Algorithm 1).

Finally, (M_{min}, R_{min}) is the pair on this curve such that it results in the minimal sum of map and reduce slots.

ALGORITHM 1: Determining the resource allocation for a Pig program

Input: Job profiles of all the jobs in $P = \{J_1, J_2, ...J_{|S_i|}\}$
$D \leftarrow$ a given deadline
$(M, R) \leftarrow$ the minimum pair of map and reduce slots obtained for P and deadline D by applying the *basic* model
Optimal execution of jobs $J_1, J_2, ...J_{|S_i|}$ based on (M, R)
Output: Resource allocation pair (M_{min}, R_{min}) for optimized P

```
$M' \leftarrow M, R' \leftarrow R$;
while $T_P^{avg}(M', R) \leq D$   // From A to B
do
    $M' \Leftarrow M' - 1$;
end
while $T_P^{avg}(M, R') \leq D$   // From A to C
do
    $R' \Leftarrow R' - 1$;
end
$M_{min} \leftarrow M, R_{min} \leftarrow R$ , $Min \leftarrow (M + R)$;
for $\hat{M} \leftarrow M' + 1$ to $M$     // Explore blue curve B to C
do
    $\hat{R} = R - 1$;
    while $T_P^{avg}(\hat{M}, \hat{R}) \leq D$ do
        $\hat{R} \Leftarrow \hat{R} - 1$;
        if $\hat{M} + \hat{R} < Min$ then
            $M_{min} \Leftarrow \hat{M}, R_{min} \Leftarrow \hat{R}, Min \leftarrow (\hat{M} + \hat{R})$;
        end
    end
end
```

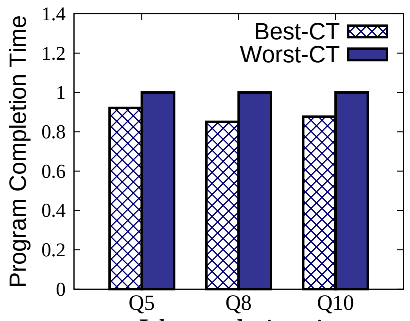
6. EVALUATION OF THE REFINED MODEL

In this section, we evaluate performance benefits of introduced optimized executions of Pig programs with concurrent jobs. Then, we assess the accuracy of the proposed *refined* performance model for completion time estimates and resource allocation decisions for Pig programs.

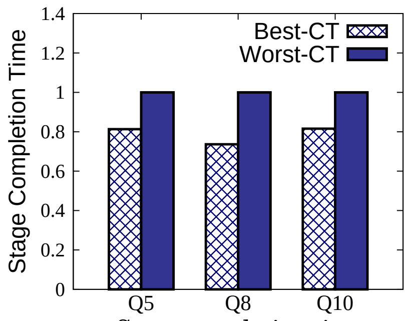
In the evaluation study, we consider the programs with concurrent jobs in two workloads: *TPC-H* and *Proxy queries*. We do not present results of *PigMix* benchmark because only 1 of 17 Pig programs included in the benchmark contains concurrent jobs. We perform the experiments using our 66 node Hadoop cluster with the same configuration as described in Section 4.

6.1. Optimal Schedule of Concurrent Jobs

Figures 12 and 13 show the scheduling impact of concurrent jobs on the program completion time for three *TPC-H queries* $Q5, Q8$, and $Q10$ and *Proxy queries* $Q1, Q2$, and $Q3$ respectively when each program in those sets is processed with **128** map and **64** reduce slots. Figures 12 (a) and 13 (a) show two extreme measurements: the best program completion time (i.e., when the optimal schedule of concurrent jobs is chosen) and the worst one (i.e., when concurrent jobs are executed in the "worst" possible order based on our estimates). For presentation purposes, the best (optimal) completion time is **normalized** with respect to the worst one. The choice of optimal schedule of concurrent jobs reduces the completion time by 10%-27% compared with the worst case ordering. Figures 12 (b) and 13 (b) show completion times of stages with con-

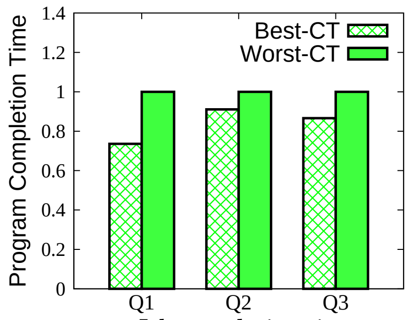


(a) Job completion times

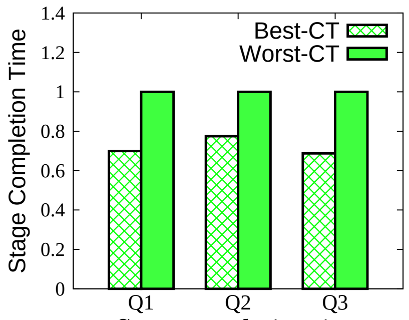


(b) Stage completion times

Fig. 12. Measured (normalized) completion times for different schedules of concurrent jobs in *TPC-H queries*.



(a) Job completion times



(b) Stage completion times

Fig. 13. Measured (normalized) completion times for different schedules of concurrent jobs in *Proxy queries*.

current jobs under different schedules for the same *TPC-H* and *Proxy queries*. The performance benefits at the stage level are even higher: they range between 20%-30%.

6.2. Predicting the Completion Time

Figure 14 shows the Pig program completion time estimates based on the earlier introduced *basic* model and and the new *refined* model for *TPC-H* and *Proxy queries* defined by the DAGs with concurrent jobs. The completion time estimates are computed using T_P^{avg} (the average of the lower and upper bounds). The programs are executed on the *experimental* dataset. Figures 14 and 15 shows the results for the case when each program is processed with **128x64** and **32x64** map and reduce slots respectively.

The completion time estimates based on the new *refined* model are significantly improved compared to the *basic* model results which are too pessimistic. In most cases (11 out of 12), the predicted completion time is within 10% of the measured ones.

6.3. Estimating Resource Allocations with Refined Model

In this set of experiments, let T denote the Pig program completion time when program P is processed with maximum available cluster resources. We set $D = 2 \cdot T$ as a completion time goal. Then we compute the required resource allocation for P when D is targeted as T_P^{avg}, i.e., the average of lower and upper bounds on the completion time.

Figures 16 (a) and 17 (a) compare the measured completion times achieved by the *TPC-H* and *Proxy's* queries respectively when they are assigned the resource allocations computed with the *basic* versus *refined* models. The completion times are **normalized** with respect to the targeted deadlines. While both models suggest sufficient resource allocations that enable the considered queries to meet their deadlines, the *basic* model significantly over-provisions resources, and the queries finish much earlier

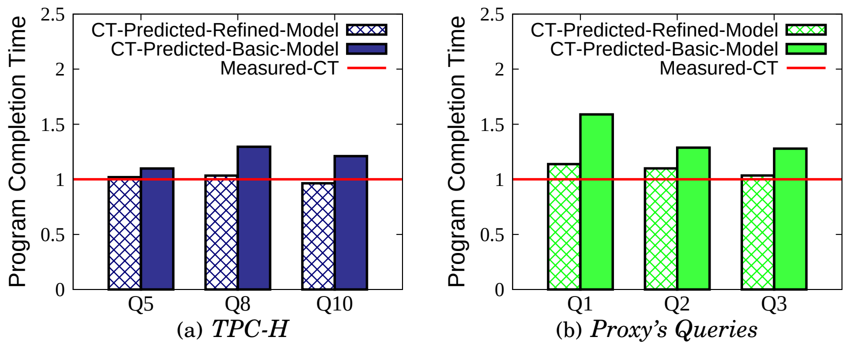


Fig. 14. Predicted completion times using *basic* vs *refined* models (**128x64** slots) with the *experimental dataset*..

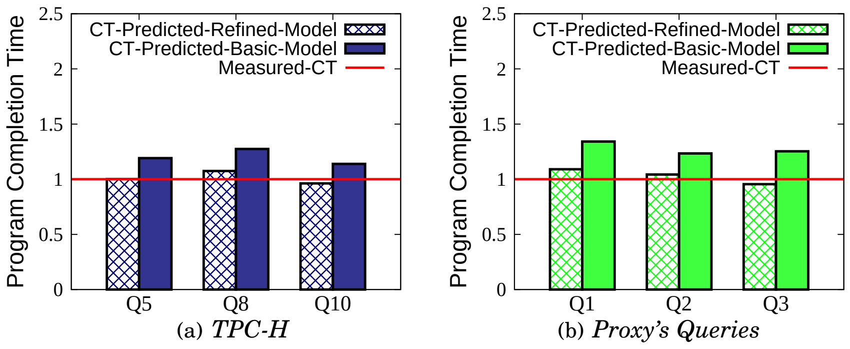


Fig. 15. Predicted completion times using *basic* vs *refined* models (**32x64** slots) with the *experimental dataset*. .

(55%-20% earlier) than given deadlines. The resource allocations computed with the *refined* model are much more accurate, and all the queries complete within 10% of the targeted deadlines.

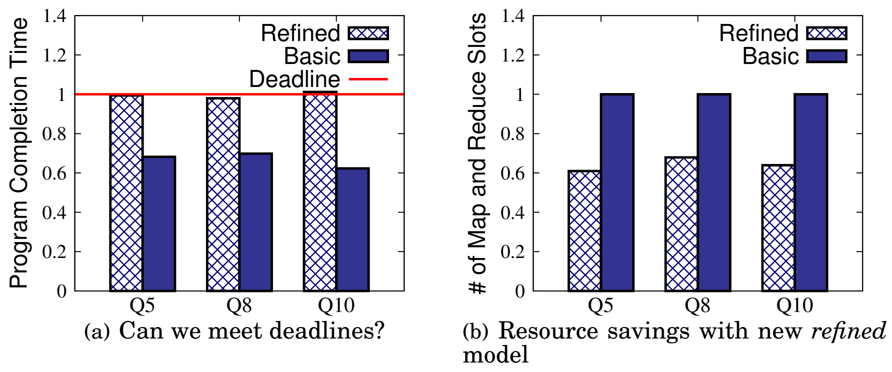
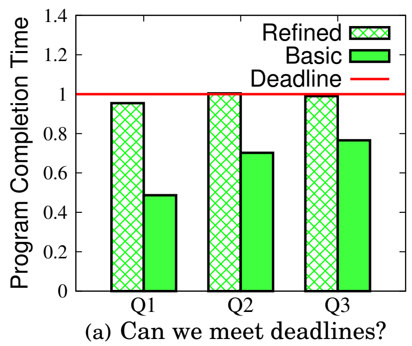
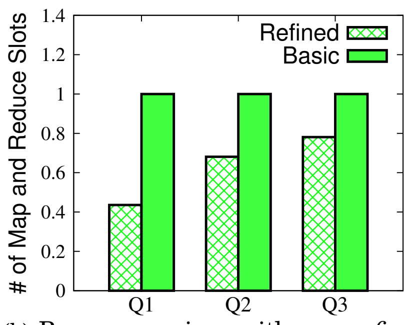


Fig. 16. *TPC-H Queries*: efficiency of resource allocations with new *refined* model.

Figures 16 (b) and 17 (b) compare the amount of resources (the sum of map and reduce slots) computed with the *basic* model versus *refined* model for *TPC-H* and *Proxy's queries* respectively. The *refined* model is able to achieve targeted deadlines with much



(a) Can we meet deadlines?



(b) Resource savings with new *refined* model

Fig. 17. *Proxy's Queries*: efficiency of resource allocations with new *refined* model.

smaller resource allocations (20%-60% smaller) compared to resource allocations suggested by the *basic* model. Therefore, the proposed optimal schedule of concurrent jobs combined with the accurate *refined* performance model lead to the efficient execution and significant resource savings for deadline-driven Pig programs.

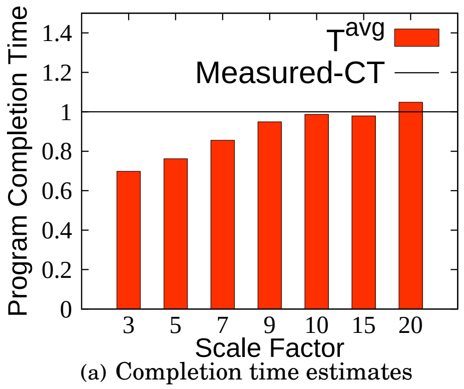
7. DISCUSSION AND FUTURE WORK

In designing our modeling framework, we exploit the fact that a typical production Hadoop (Pig) job is executed routinely on new data. We take advantage of this observation, and for a periodic Pig program, we automatically build its jobs' profiles from the past execution. These extracted job profiles are used for **future** predictions when this Pig program is executed on new data. The question is how sensitive the model and the prediction results to a given training data (i.e. to the extracted job profile from the last job execution). If the amount of new data is close in size (e.g., $\pm 20\%$) to the data processed in the past (when the Pig jobs' profiles were built) then the proposed models work well and they are accurate. Most of our predictions are within 10%-15% of the measured results.

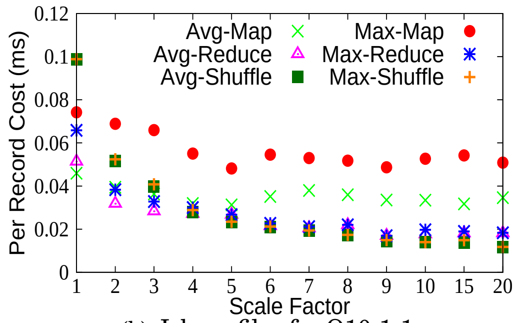
However, if the Pig program processes a significantly different amount of data (e.g., twice as much) compared to the past execution when the job profiles were built – how does it impact the accuracy of our models and their predictions?

In our profiling approach, we estimate processing costs per record during different execution stages: map, shuffle/sort, and reduce phases. This normalized cost is used to project the map/reduce task duration when these tasks need to process the increased (or decreased) amounts of data (records). Figure 18 (a) shows a predicted completion time (normalized with respect to the measured time) of *TPC-H Q10* with different scale factors. Our prediction is based on the job profiles extracted from *TPC-H Q10* with the scale factor 9.

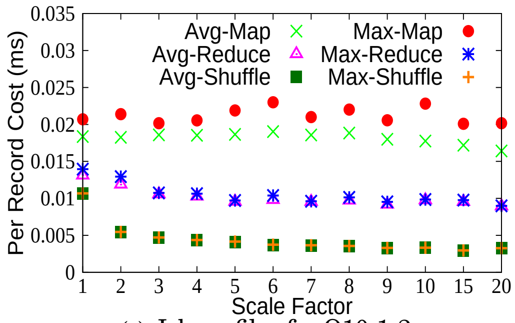
Using these job profiles we predict T^{avg} for *TPC-H Q10* based on different scale factors. The predicted results are accurate for scale factors 10,15, 20, and still acceptable for scale factor 7. The prediction errors for scale factors 3 and 5 are significantly higher. To understand the logic behind these results, we analyze *TPC-H Q10* processing in more detail (see its DAG in Figure 2 (c)). Our measurements reveal that the first stage is responsible for 40% of the overall execution time. Therefore, analyzing the profiles of two concurrent jobs $J1$ and $J2$ of this stage will be very useful. Figures 18 (b) and (c) show the processing costs per record during different execution phases of $J1$ and $J2$ respectively. Apparently, the job profiles are quite stable starting at the scale factor 8 and up. It explains why the completion time predictions for these scale factors are accurate. However, for small scale factors, the processing costs per record, especially in reduce



(a) Completion time estimates



(b) Job profiles for Q10-1-1



(c) Job profiles for Q10-1-2

Fig. 18. Completion time estimates for *TPC-H Q10* with different input data size (scale factor).

and shuffle phases, are significantly higher. There is an overhead associated with a task execution and it contributes a significant portion in the task duration when processing a small set of records. The overhead impact is significantly diminished when processing a larger dataset, and the cost per record becomes more representative of the actual processing cost.

In our future work, we plan to investigate these overheads in more detail to take them into account for computing the per record cost and in scaling the job profiles for different amount of input data. The other interesting alternative is to derive (and interpolate) the per record cost empirically using the available set of past runs of the Pig program on datasets of different sizes.

The other related issue is the impact of resource contention on the Pig program completion time when multiple Pig programs or MapReduce jobs are executed simultaneously. Even when a Pig program is executed on the Hadoop cluster in isolation there is an inevitable resource contention because there are multiple slots at each worker node, and the different tasks of the same job scheduled on these slots compete for the shared node resources. These tasks compete for the disk i/o and the network resources in a similar way as the tasks of different jobs. When the designed performance models become a part of the SLO-based scheduler [Verma et al. 2011a], there is an additional scheduler feature that can increase or decrease the number of slots allocated to a job based on the expected progress of this job. This additional scheduler feature aims to compensate the possible model inaccuracy due to resource contention or unforeseen executional runtime non-determinism, and in such a way improve the overall scheduler outcome.

8. 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. [Polo et al. 2010] 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 [Wolf et al. 2010] 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 [Verma et al. 2011a] 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. The shortcoming of this work is that it does not have scaling factors to adjust the extracted profile when the job is executed on a different size dataset. In the follow up work [Verma et al. 2011b], a more general profiling approach that is based on regression technique for deriving the scaling factors is proposed. However, these models apply to a single MapReduce job.

Tian and Chen [Tian and Chen] 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 task is partitioned in 4 functions: read a block, map function processing of the block, partition and sort of the processed data, and the combiner step (if it is used). The reduce task is decomposed in 4 functions as well. The authors use a *linear regression technique* to approximate the cost (duration) of each function. These functions are used for predicting the larger dataset processing. There are a few simplifying assumptions, e.g., a single wave in the reduce stage. 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.

There is an interesting group of papers that design a detailed job profiling approach for pursuing a different goal: to optimize the configuration parameters of both a Hadoop cluster and a given MapReduce job (or workflow of jobs) for achieving the improved completion time. *Starfish* project [Herodotou et al. 2011; Herodotou and Babu 2011] 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. Such a 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. In our work, we propose complementary optimizations based on optimal scheduling of concurrent jobs within the DAG to minimize overall completion time.

Kambatla et al [Kambatla et al. 2009] propose a different approach for optimizing the Hadoop configuration parameters (number of map and reduce slots per node) to improve MapReduce program performance. A signature-based (fingerprint-based) approach is used to predict the performance of a new MapReduce program using a set of already studied programs. The ideas presented in the paper are interesting but it is a position paper that does not provide enough details and is lacking the extended evaluation of the approach.

Ganapathi et al. [Ganapathi et al. 2010] use Kernel Canonical Correlation Analysis (KCCA) to predict the performance of *Hive queries* according to job feature vectors that include the number and location of map (reduce) tasks and data characteristics such as bytes read locally and from HDFS. The authors build the KCCA model through detecting the statistical correlation of the job features and performance features using a large training set (5000 of Hive queries). Our method does not require such a large training set. We extract a Pig program profile from the latest past run of this program, and use this profile for predicting the future execution of this Pig program on the new dataset. The accuracy of KCCA is not very clear, because the completion time of any Pig program (or Hive query, or any MapReduce job) depends on the amount of allocated resources (the number of map and reduce slots) to this program. The amount of allocated resources is not explicitly discussed in the paper [Ganapathi et al. 2010], and the authors do not provide details on the amount of allocated resource used by the Hive queries in the training set. It is not clear whether this model holds for different resource allocation to these queries, and how often this model needs to be retrained for a changed workload.

CoScan [Wang et al. 2011] 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. [Morton et al. 2010a] propose *ParaTimer*: the progress estimator for parallel queries expressed as Pig scripts [Gates et al. 2009]. In their earlier work [Morton et al. 2010b], 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 (under the specific FIFO scheduler execution), and therefore identifying all the pipelines (sequences of MapReduce jobs) in the query. The approach relies on a simplified assumption that map (reduce) tasks of the same job have the same duration. The proposed technique estimates the remaining query execution time by comparing the number of tuples already processed to the number of remaining tuples. This heuristic is only effective when there is a constant tuple processing rate during the query execution. However, this assumption is typically does not hold in practice due to job and task execution non-determinism and other runtime effects.

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. The proposed approach does not enable the solution of the inverse 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 Pig program with a given (*soft*) deadline.

9. CONCLUSION

Design of new job profiling tools and performance models for MapReduce environments has been an active research topic in industry and academia during past few years. Most of these efforts were driven by design of new schedulers to satisfy the job specific goals and to improve cluster resource management. In our work, we have introduced a novel performance modeling framework for processing Pig programs with deadlines. Our job profiling technique is not intrusive, it does not require any modifications or instrumentation of either the application or the underlying Hadoop/Pig execution engines. The proposed approach enables automated SLO-driven resource sizing and provisioning of complex workflows defined by the DAGs of MapReduce jobs. Moreover, our approach offers an optimized scheduling of concurrent jobs within a DAG that allows to significantly reduce the overall completion time.

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.

We validated the accuracy of our approach and performance models in the 66-node Hadoop cluster. In our testbed, the network was not a bottleneck. Typically, service providers tend to over provision network resources to avoid undesirable side effects of network contention. However, in the large Hadoop clusters the network overprovisioning is more difficult and expensive. It is an interesting future work whether the network contention factor can be introduced in the proposed performance models.

REFERENCES

http://www.tpc.org/tpch/, 2008. TPC Benchmark H (Decision Support), Version 2.8.0, Transaction Processing Performance Council (TPC). (http://www.tpc.org/tpch/, 2008). http://www.tpc.org/tpch/

Apache. http://wiki.apache.org/pig/PigMix, 2010. PigMix Benchmark. (http://wiki.apache.org/pig/PigMix, 2010). http://wiki.apache.org/pig/PigMix

R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. 2008. Easy and Efficient Parallel Processing of Massive Data Sets. *Proc. of the VLDB Endowment* 1, 2 (2008).

J. Dean and S. Ghemawat. 2008. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM* 51, 1 (2008).

A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson. 2010. Statistics-driven workload modeling for the cloud. In *Proc. of 5th International Workshop on Self Managing Database Systems (SMDB)*.

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

H. Herodotou and S. Babu. 2011. Profiling, What-if Analysis, and Costbased Optimization of MapReduce Programs.. In *Proc. of the VLDB Endowment, Vol. 4, No. 11*.

H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics.. In *Proc. of 5th Conf. on Innovative Data Systems Research (CIDR)*.

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

S.M. Johnson. 1954. Optimal Two- and Three-Stage Production Schedules with Setup Times Included. *Naval Res. Log. Quart.* (1954).

K. Kambatla, A. Pathak, and H. Pucha. 2009. Towards optimizing hadoop provisioning in the cloud. In *Proc. of the First Workshop on Hot Topics in Cloud Computing*.

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

K. Morton, A. Friesen, M. Balazinska, and D. Grossman. 2010b. Estimating the progress of MapReduce pipelines. In *Proc. of ICDE*.

J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley. 2010. Performance-Driven Task Co-Scheduling for MapReduce Environments. In *Proc. of the 12th IEEE/IFIP Network Operations and Management Symposium*.

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

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)*.

Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011a. ARIA: Automatic Resource Inference and Allocation for MapReduce Environments. *Proc. of the 8th ACM International Conference on Autonomic Computing (ICAC'2011)* (2011).

Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011b. SLO-Driven Right-Sizing and Resource Provisioning of MapReduce Jobs. *Proc. of the 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS)* (2011).

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

Joel Wolf, Deepak Rajan, Kirsten Hildrum, Rohit Khandekar, Vibhore Kumar, Sujay Parekh, Kun-Lung Wu, and Andrey Balmin. 2010. FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads. *Proc. of the 11th ACM/IFIP/USENIX Middleware Conference* (2010).