AutoTune: Optimizing Execution Concurrency and Resource Usage in MapReduce Workflows*

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

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

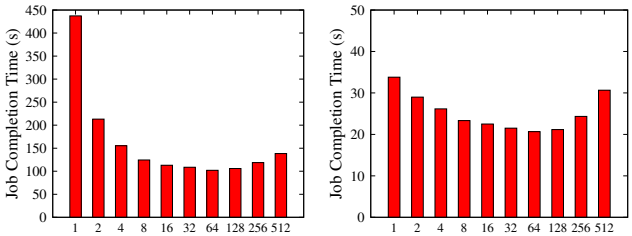
Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Abstract

An increasing number of MapReduce applications are written using high-level SQL-like abstractions on top of MapReduce engines. Such programs are translated into MapReduce workflows where the output of one job becomes the input of the next job in a workflow. A user must specify the number of reduce tasks for each MapReduce job in a workflow. The reduce task setting may have a significant impact on the execution concurrency, processing efficiency, and the completion time of the worklflow. In this work, we outline an automated performance evaluation framework, called *AutoTune*, for guiding the user efforts of tuning the reduce task settings in MapReduce sequential workflows while achieving performance objectives. We evaluate performance benefits of the proposed framework using a set of realistic MapReduce applications: *TPC-H* queries and custom programs mining a collection of enterprise web proxy logs.

1 Introduction

Many companies are embracing MapReduce environments for advanced data analytics over large datasets. Optimizing the execution efficiency of these applications is a challenging problem that requires the user experience and expertize. Pig [4] and Hive [10] frameworks offer high-level SQL-like languages and processing systems 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) and *workflows* of MapReduce jobs. Currently, a user must specify the number of reduce tasks for each MapReduce job (the default setting is 1 reduce task). Determining the right number of reduce tasks is non-trivial: it depends on the input sizes of the job, on the Hadoop cluster size, and the amount of resources available for processing this job. In the MapReduce workflow, two sequential jobs are data dependent: the output of one job becomes the input of the next job, and therefore, the number of reduce tasks in the previous job defines the number (and size) of input files of the next job, and may affect its performance and processing efficiency in unexpected ways. To demonstrate these issues we use a *Grep* program provided with the Hadoop distribution. This program consists of a workflow with two sequential jobs: the first job ($J1$) searches for strings with the user-specified patterns and counts them for each matched pattern. The second job ($J2$) reads the output of the first job and sorts the patterns according to their appearance frequencies. We execute this program with 15 GB of input data on a Hadoop cluster with 64 worker nodes. Each node is configured with 2 map and 1 reduce slots, i.e., with 128 map and 64 reduce slots overall. Figure 1 shows the execution times of both jobs $J1$ and $J2$ as we vary the number of reduce tasks in $J1$. (The number of reduce task for $J2$ is fixed to 1).



(a) Job $J1$: Search (b) Job $J2$: Sort

Figure 1: A Motivating Example.

Figure 1 (a) shows that the $J1$ execution time significantly depends on the number of reduce tasks. A low number of reduce tasks limits the job execution concurrency and leads to an increased job completion time. A high number of reduce tasks improves the execution parallelism and the job processing time, but at some point (e.g., 512 reduce tasks), it leads to an increased overhead and increased job processing time.

As the outputs generated by $J1$ become inputs of $J2$, the number of output files and their sizes may have a significant impact on the performance of J_2. Figure 1 (b) shows how the reduce task settings of $J1$ impact the completion time of $J2$.

In this work, we outline a novel performance evaluation framework, called *AutoTune*, that automates the user efforts of tuning the numbers of reduce tasks along the MapReduce workflow. It consist of the following key components:

- *The ensemble of performance models* that orchestrates

*This work was mostly done uring Z. Zhang's internship at HP Labs. Prof. B. T. Loo a nd Z. Zhang are supported in part by NSF grants CNS-1117185 and CNS-0845552.

the prediction of the workflow completion time at the cluster and application levels.

- *Optimization strategies* that are used for determining the numbers of reduce tasks along the jobs in the MapReduce workflow for achieving the performance objectives and for analyzing the performance trade-offs.

We validate the accuracy, efficiency, and performance benefits of the proposed framework using a set of realistic MapReduce applications executed on 66-nodes Hadoop cluster. This set includes TPC-H queries and custom programs mining a collection of enterprise web proxy logs. Our case study shows that the proposed ensemble of models accurately predicts workflow completion time. Moreover, the proposed framework enables users to analyze the efficiency trade-offs. Our experiments show that in many cases, by allowing 5%-10% increase in the workflow completion time one can gain 40%-90% of resource usage savings. The ability to optimize the amount of resources used by the programs enables efficient workload management in the cluster.

This paper is organized as follows. Section 2 presents the problem definition and outlines our solution. Sections **??**-3 describe the ensemble of performance models and optimization strategies. Section 4 evaluates the framework accuracy and effectiveness of optimization strategies. Section 5 outlines related work. Section 6 presents conclusion and future work directions.

2 *AutoTune* Solution Outline

Currently, a user must specify the number of reduce tasks for each MapReduce job in a workflow (the default setting is 1 reduce task). The reduce task setting defines the number of paraller tasks that are created for processing data at the reduce stage and the amount of data which is processed and written by each task. As a result, the reduce tasks settings impact the efficiency of the map stage processing in the next job. If too many small output files created it leads to a higher processing overhead and a higher number of map slots is needed for these files processing. Our **main goal** is to determine the reduce tasks settings that *optimize the workflow overall completion time*. Typically, the Hadoop cluster is shared by multiple users and their jobs are scheduled with Fair or Capacity Schedulers. Under these schedulers the cluster resources are partitioned into pools with separate queues and priorities for each pool. The unused cluster resources can be allocated to any pool(s) with jobs that can utilize these resources. Therefore, an **additional goal** is to *minimize the workflow resource usage* for achieving this optimized time. Often nearly optimal completion time can be achieved with a significantly smaller amount of resources (compare the outcome of 32 and 64 reduce task settings in Figure 1).

AutoTune solution relies on the following *Pairwise Optimization Theorem*: the optimization problem of the entire workflow can be efficiently solved through the optimization problem of the pairs of its sequential jobs.

Proof: Figure 2 shows a workflow that consists of three sequential jobs: J_1, J_2, and J_3. To optimize the workflow completion time we need to tune the reduce task settings in jobs J_1, J_2, and J_3. A *question* to answer is whether the choice of reduce task setting in job J_1 impacts the choice of reduce task setting in job J_2, etc.

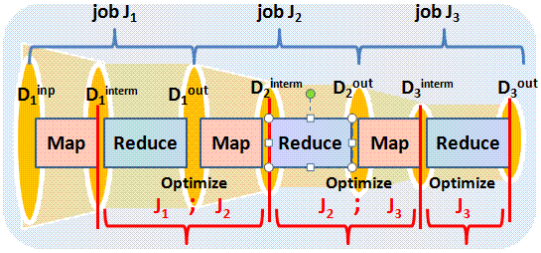


Figure 2: Example workflow with 3 sequential jobs

A *critical observation* here is that *the size of overall data* generated between the map and reduce stages of the same job and between two sequential jobs *does not depend on the reduce task settings* of these jobs. For example, the overall amount of output data D_1^{out} of job J_1 does not depend on the number of reduce tasks in J_1. It is defined by the size and properties of D_1^{interm}, and the semantics of J_1's reduce function. Similarly, the amount of D_2^{interm} is defined by the size of D_1^{out}, properties of this data, and the semantics of J_2's map function. Again, the size of D_2^{interm} does not depend on the number of reduce tasks in J_1. Therefore the amount of intermediate data generated by the map stage of J_2 is the same (i.e., *invariant*) for different settings of reduce tasks in the previous job J_1. It means that the choice of an appropriate number of reduce tasks in job J_2 does not depend on the choice of reduce task setting of job J_1. It is primarily driven by an optimized execution of the next pair of jobs J_2 and J_3. Finally, tuning the reduce task setting in J_3 is driven by optimizing its own completion time.■

In such a way, the optimization problem of the entire workflow can be efficiently solved through the **optimization problem of the pairs of its sequential jobs**. Therefore, for two sequential jobs J_1 and J_2, we need to design a model that evaluates the execution times of J_1's reduce stage and J_2's map stage as a function of a number of reduce tasks in J_1. Such a model will enable us to iterate through a range of reduce tasks' parameters and identify a parameter that leads to the minimized completion time of these jobs and evaluate the amount of utilized resources (both reduce and map slots used over time).

While there were quite a few research efforts to design a model for predicting the completion time of a MapReduce job [6, 5, 11, 12, 13], it still remains a challenging research problem. The **main challenge** is to estimate the durations of map and reduce tasks (and the entire job) when these tasks process different amount of data (compared to past job runs).

Some earlier modeling efforts for predicting the job completion time analyze map and reduce task durations [12] from the past job runs, and then derive some scaling factors for task

execution times when the original MapReduce application is applied for processing a larger dataset [13, 11]. Some other efforts [6, 5, 11] aim to perform a more detailed (and more expensive) job profiling and time prediction at a level of phases that comprise the execution of map and reduce tasks.

In this work, we pursue a new approach for designing a MapReduce performance model that can efficiently predict the completion time of a MapReduce application for processing different given datasets as a function of allocated resources. It combines the useful rationale of the detailed phase profiling method [6] for estimating durations of map/reduce tasks with fast and practical analytical model designed in [12]. However, our new framework proposes a very *different approach* to estimate the execution times of these job phases. We observe that the executions of map and reduce tasks consist of specific, well-defined data processing phases. Only map and reduce functions are custom and their computations are user-defined for different MapReduce jobs. The executions of the remaining phases are *generic*, i.e., strictly regulated and defined by the Hadoop processing framework. The execution time of each generic step depends on the amount of data processed by the phase and the performance of underlying Hadoop cluster. In the earlier papers [6, 5, 11], profiling is done for all the phases (including the generic ones) for each application separately. Then these measurements are used for predicting a job completion time.

In our work, we design *a set of parameterizable microbenchmarks* [16] to measure generic phases and to derive *a platform performance model* of a given Hadoop cluster. We distinguish *five* phases of the map task execution and *three* phases of the reduce task execution as shown in Figure 3.

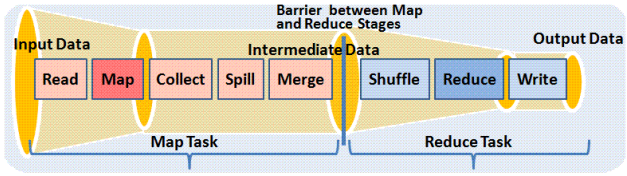


Figure 3: MapReduce Processing Pipeline.

We concentrate on profiling of generic (non-customized) phases of the *MapReduce processing pipeline* (opposite to phase profiling of specific MapReduce jobs). By running a set of diverse benchmarks on a given Hadoop cluster we collect a useful training set (that we call a *platform profile*) that characterizes the execution time of different phases while processing different amounts of data. This profiling can be done in a small test cluster with the same hardware and configuration as the production cluster. Using the created training set and a robust linear regression we derive a *platform performance model* $\mathscr{M}_{platform} = (\mathscr{M}_{read}, \mathscr{M}_{collect}, \mathscr{M}_{spill}, \mathscr{M}_{merge}, \mathscr{M}_{shuffle}, \mathscr{M}_{write})$ that estimates each phase duration as a function of processed data:

$$T^J_{phase} = \mathscr{M}_{phase}(Data^J_{phase}), \quad (1)$$

where $phase \in \{read, collect, spill, merge, shuffle, write\}$.

The proposed evaluation framework aims to **divide** *i)* the performance characterization of the underlying Hadoop cluster and *ii)* the extraction of specific performance properties of different MapReduce applications. It aims to derive **once** an accurate performance model of Hadoop's generic execution phases as a function of processed data, and then **reuse** this model for characterizing performance of generic phases across different applications (with different job profiles).

For profiling map and reduce phases (user-defined map and reduce functions) of production MapReduce jobs we apply our alternative profiling tool based on *BTrace* approach [2]. It can be applied to jobs in the production cluster. Since we only profile map and reduce phase execution – the overhead is small.

Once we approximate the execution times of map and reduce tasks, we could model the completion time of a single job by applying the analytical model designed in ARIA [12]. The proposed performance model utilizes the knowledge about average and maximum of map/reduce task durations for computing the lower and upper bounds on the job completion time as a function of allocated map and reduce slots. Equation 2 shows the lower-bound on the job completion time:

$$T^{low}_J = \frac{N^J_M \cdot M^J_{avg}}{S^J_M} + \frac{N^J_R \cdot R^J_{avg}}{S^J_R} \quad (2)$$

where M^J_{avg} (R^J_{avg}) represent the average map (reduce) task duration, N^J_M (N^J_R) denote the map (reduce) task number and S^J_M (S^J_R) reflect the number of map (reduce) slots for processing the job. The computation of the upper bound on the job completion time is slightly different (see [12] for details: the formula involves the estimates of maximum map/reduce task durations). As it is shown in [12], the average of lower and upper bounds serves as a good prediction of the job completion time (it is within 10% of the measured one).

3 Two Optimization Strategies

According to *Pairwise Optimization Theorem* the optimization problem of reduce task settings for a given workflow $W = \{J_1, ..., J_n\}$ can be efficiently solved via the *optimization problem of the pairs of its sequential jobs*. Therefore, for any two sequential jobs (J_i, J_{i+1}), where $i = 1, ..., n-1$, we need to evaluate the execution times of J_i's reduce stage and J_{i+1}'s map stage as a function of the number of reduce tasks $N^{J_i}_R$ in J_i (see the related illustration in Figure 2, Section 2). Let us denote this execution time as $T_{i,i+1}(N^{J_i}_R)$.

By iterating through the number of reduce tasks in J_i we can find the reduce task setting $N^{J_i,min}_R$ that results in the minimal completion time $T^{min}_{i,i+1}$ for the pair (J_i, J_{i+1}), i.e., $T^{min}_{i,i+1} = T_{i,i+1}(N^{J_i,min}_R)$. By determining the reduce task settings s for all the job pairs, i.e., $s^{min} = \{N^{J_1,min}_R, ..., N^{J_n,min}_R\}$, we can determine the minimal workflow completion time $T_W(s^{min})$. *AutoTune* can be used with Hadoop Fair Scheduler or Capacity Scheduler and multiple jobs executed on a

cluster. Both schedulers allow configuring different size resource pools each running jobs in the FIFO manner. Note, that the proposed approach for finding the reduce task setting that minimizes the workflow completion time can be applied to a different amount of available resources, e.g., the entire cluster or a fraction of available cluster resources. Therefore, the optimized workflow execution can be constructed for any size resource pool managed (available) in a Hadoop cluster.

We aim to design the optimization strategy that enables a user to analyze the possible trade-offs, such as workflow performance versus its resource usage. We aim to *answer the following question*: if the performance goal allows a specified increase of the minimal workflow completion time $T_W(s^{min})$, e.g., by 10%, then what is the resource usage under this workflow execution compared to $R_W(s^{min})$?

We define the resource usage $R_{i,i+1}(N_R^{J_i})$ for a sequential job pair (J_i, J_{i+1}) executed with the number of reduce tasks $N_R^{J_i}$ in job J_i as follows:

$$R_{i,i+1}(N_R^{J_i}) = T_{R\_task}^{J_i} \times N_R^{J_i} + T_{M\_task}^{J_{i+1}} \times N_M^{J_{i+1}}$$

where $N_M^{J_{i+1}}$ represent the number of map tasks of job J_{i+1}, and $T_{R\_task}^{J_i}$ and $T_{M\_task}^{J_{i+1}}$ represent the average execution time of reduce and map tasks of J_i and J_{i+1} respectively. The resource usage for the entire MapReduce workflow is defined as the sum of resource usage for each job within the workflow.

Table 1 summarizes the notations that we use for defining the optimization strategies below.

Table 1: Notation Summary

$T_{i,i+1}(N_R^{J_i})$	Completion time of (J_i, J_{i+1}) with $N_R^{J_i}$ reduce tasks
$R_{i,i+1}(N_R^{J_i})$	Resource usage of pair (J_i, J_{i+1}) with $N_R^{J_i}$ reduce tasks
$T_W(s)$	Completion time of the entire workflow W with setting s
$R_W(s)$	Resource usage of the entire workflow W with setting s
$T_{i,i+1}^{min}$	Minimal completion time of a job pair (J_i, J_{i+1})
$N_R^{J_i,min}$	Number of reduce tasks in J_i that leads to $T_{i,i+1}^{min}$
w_increase	Allowed increase of the min workflow completion time
$N_R^{J_i,incr}$	Number of reduce tasks in J_i to meet the increased time

The first algorithm is based on the *local optimization*. The user specifies the allowed increase *w_increase* of the minimal workflow completion time $T_W(s^{min})$. Our goal is to compute the new workflow reduce task settings that allow achieving this increased completion time. To accomplish this goal, a straightforward approach is to apply the user-defined *w_increase* to the minimal completion time $T_{i,i+1}^{min}$ of each pair of sequential jobs (J_i, J_{i+1}), and then determine the corresponding number of reduce tasks in J_i. The pseudo-code defining this strategy is shown in Algorithm 1. The completion time of each job pair is locally increased (line 2), and then the reduce task settings are computed (lines 4-6).

While this local optimization strategy is simple to implement, there could be additional resource savings achieved if we consider a *global optimization*. Intuitively, the resource usage for job pairs along the workflow might be quite different depending on the job characteristics. Therefore, we could identify the job pairs with the highest resource savings (gains) for their increased completion times. The pseudo-code defining this global optimization strategy is shown in Algorithm 2.

Algorithm 1 Local optimization strategy for deriving workflow reduce tasks' settings

1: **for** $i \leftarrow 1$ **to** n **do**
2: $\quad T_{i,i+1}^{incr} = T_{i,i+1}^{min} \times (1 + w\_increase)$
3: $\quad N_R^{J_i,cur} \leftarrow N_R^{J_i,min}$
4: $\quad$ **while** $T_{i,i+1}(N_{R_i}^{J_i,cur}) < T_{i,i+1}^{incr}$ **do**
5: $\quad\quad N_R^{J_i,cur} \leftarrow N_R^{J_i,cur} - 1$
6: $\quad$ **end while**
7: $\quad N_R^{J_i,incr} \leftarrow N_R^{J_i,cur}$
8: **end for**

Algorithm 2 Global optimization strategy for deriving workflow reduce tasks' settings

1: $s^{cur} = s^{min} = \{N_R^{J_1,min}, ..., N_R^{J_n,min}\}$
2: $T_{w\_incr} = T_W(s^{min}) \times (1 + w\_increase)$
3: **for** $i \leftarrow 1$ **to** n **do**
4: $\quad N_R^{J_i,incr} \leftarrow N_R^{J_i,min}$
5: **end for**
6: **while** true **do**
7: $\quad bestJob = -1, \quad maxGain = 0$
8: $\quad$ **for** $i \leftarrow 1$ **to** n **do**
9: $\quad\quad N_R^{J_i,tmp} \leftarrow N_R^{J_i,incr} - 1$
10: $\quad\quad s^{tmp} = s^{cur} \cup \{N_R^{J_i,tmp}\} - \{N_R^{J_i,incr}\}$
11: $\quad\quad$ **if** $T_W(s^{tmp}) \leq T_{w\_incr}$ **then**
12: $\quad\quad\quad Gain = \frac{R_W(s^{min}) - R_W(s^{tmp})}{T_W(s^{tmp}) - T_W(s^{min})}$
13: $\quad\quad\quad$ **if** $Gain > MaxGain$ **then**
14: $\quad\quad\quad\quad maxGain \leftarrow Gain, \quad bestJob \leftarrow i$
15: $\quad\quad\quad$ **end if**
16: $\quad\quad$ **end if**
17: $\quad$ **end for**
18: $\quad$ **if** $bestJob = -1$ **then**
19: $\quad\quad$ break
20: $\quad$ **else**
21: $\quad\quad N_R^{bestJob,incr} \leftarrow N_R^{bestJob,incr} - 1$
22: $\quad$ **end if**
23: **end while**

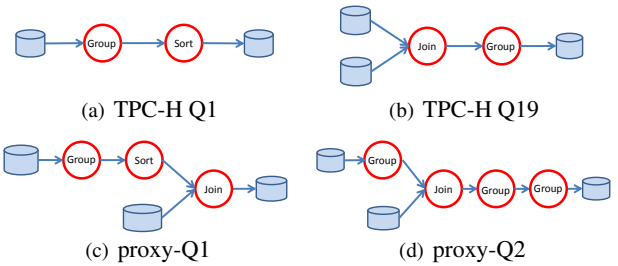
First, we apply the user-specified *w_increase* to determine the targeted completion time T_{w_incr} (line 2). The initial number of reduce task for each job J_i is set to $N_R^{J_i,min}$ (lines 3-5), and then we go through the iteration that at each round estimates the *gain* we can get by decreasing the number of reduce tasks by one for each job J_i. We aim to identify the job that has the smallest response time increase with the decreased amount of reduce tasks while satisfying the targeted workflow completion time (lines 8-17). We pick the job which brings the largest gain and decrease its reduce task setting by 1 (line 21). Then the iteration repeats until the number of reduce tasks in any job cannot be further decreased because it would cause a violation of the targeted workflow completion time T_{w_incr} (line 11).

4 Evaluation

Experimental Testbed and Workloads. All experiments are performed on 66 HP DL145 G3 machines. Each machine has four AMD 2.39GHz cores, 8 GB RAM and two 160 GB 7.2K rpm SATA hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We use Hadoop 1.0.0 and Pig-0.7.0 with two machines dedicated to*JobTracker* and *NameNode*, and remaining 64 machines as workers. Each worker is configured with 2 map and 2 reduce slots. The HDFS 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.

To validate the accuracy, effectiveness, and performance benefits of the proposed framework, we use queries from TPC-H benchmark and custom queries mining a collection of web proxy logs. TPC-H [3] is a standard database benchmark for decision-support workloads. It comes with a data generator for creating the test database. The input dataset size is controlled by the scaling factor: the scaling factor of 1 generates 1 GB input dataset. Our second dataset contains 6 months access logs of the enterprise web proxy during 2011-2012 years.

TPC-H and proxy queries are implemented using Pig [4]. Queries are translated into *sequential* MapReduce workflows that are graphically represented in Fig. 4.



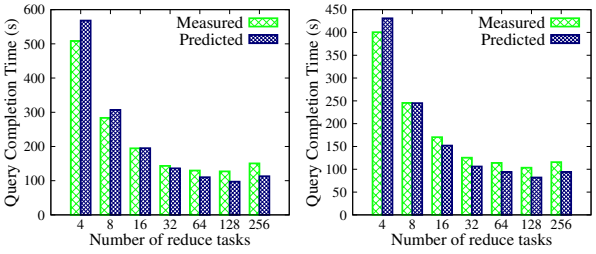
(a) TPC-H Q1 (b) TPC-H Q19

(c) proxy-Q1 (d) proxy-Q2

Figure 4: Workflows for TPC-H and Proxy queries.

AutoTune Performance Benefits. Since it is infeasible to validate optimal settings by testbed executions (unless we exhaustively execute the programs with all possible settings), we evaluate the models' accuracy to justify the optimal settings procedure.

We execute two queries *TPC-H Q1* and *TPC-H Q19* with the total input size of 10 GB in our 66-node Hadoop cluster. Figure 5 shows measured and predicted query completion times for a varied number of reduce tasks in the first job of both workflows (the number of reduce tasks for the second job is fixed in these experiments). First of all, results presented in Figure 5 reflect a good quality of our models: the difference between measured and predicted completion times for most of the experiments is less than 10%. Moreover, the predicted completion times accurately reflect a similar trend observed in measured completion times of the studied workflows as a function of the reduce task configuration. These experiments demonstrate that there is a significant difference (up to 4-5 times) in the workflow completion times depending on the reduce task settings.



(a) *TPC-H Q1* (b) *TPC-H Q19*

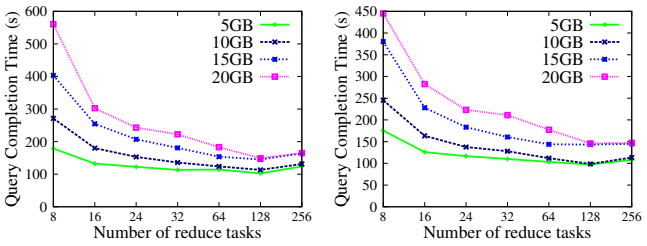
Figure 5: Model validation for *TPC-H Q1* and *TPC-H Q19*.

Figure 5 shows that the query completion time decreases with the increased number of reduce tasks (because it leads to a higher concurrency degree and a smaller amount of data processed by each task). However, at some point job settings with a high number of reduce tasks (e.g., 256) may have a negative effect due to higher overheads and higher resource allocation required to process such a job.

Another interesting observation from the results in Figure 5 is that under two settings with a number of reduce tasks equal to 64 and 128 the workflow completion times are very similar while the number of required reduce slots for a job execution increases twice. As shown later in this section, *AutoTune* enables the user to identify useful trade-offs in achieving the optimized workflow completion time while minimizing the amount of resources required for a workflow execution.

Why Not Use Best Practices? There is a list of best practices [1] that offers useful guidelines in determining the appropriate configuration settings. The widely used *rule of thumb* suggests to set the number of reduce tasks to 90% of all available resources (reduce slots) in the cluster. Intuitively, this maximizes the concurrency degree in job executions while leaving some "room" for recovering from the failures. This approach may work under the FIFO scheduler when all the cluster resource are (eventually) available to the next scheduled job. This guideline does not work well when the Hadoop cluster is shared by multiple users, and their jobs are scheduled with Fair or Capacity Schedulers. Moreover, the rule of thumb suggests the same number of reduce tasks for all applications without taking into account the amount of input data for processing in these jobs.

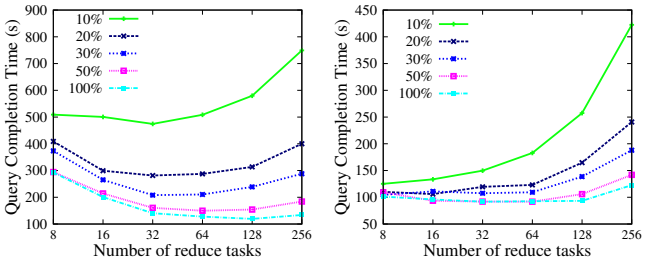
To illustrate these issues, Figure 6 shows the impact of the number of reduce tasks on the measured query completion time for executing *TPC-H Q1* and *TPC-H Q19* with different input dataset sizes. The *rule of thumb* suggests to use 115 reduce tasks (128*0.9=115). However, as we can see from the results in Figure 6 (a), for dataset sizes of 10 GB and 15 GB the same performance could be achieved with 50% of the suggested resources. The resource savings are even higher for *TPC-H Q1* with 5 GB input size: it can achieve the nearly optimal performance by using only 24 reduce tasks (this represents 80% savings against the *rule of thumb* setting). The results for *TPC-H Q19* show similar trends and conclusion.



(a) *TPC-H Q1* (b) *TPC-H Q19*

Figure 6: Effect of reduce task settings for processing the same job with different input dataset sizes (measured results).

In addition, Figure 7 shows the effect of reduce task settings on the measured completion time of *TPC-H Q1* query when only a fraction of resources (both map and reduce slots) is available for the job execution. Figures 7 (a) and (b) show

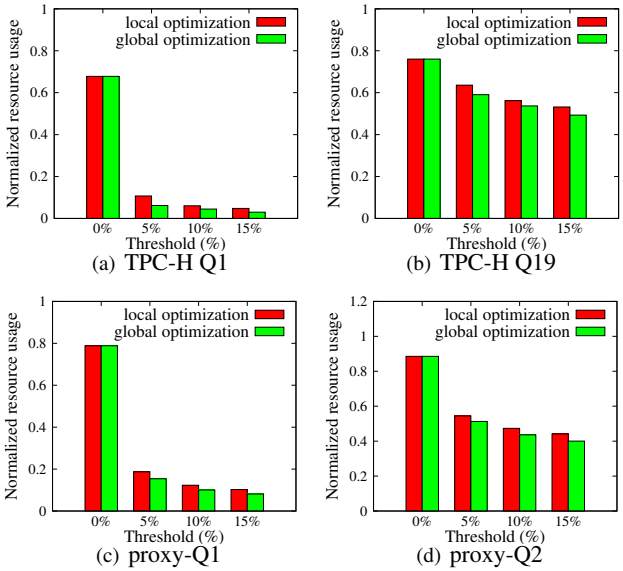


(a) input size 10 GB (b) input size 1 GB

Figure 7: Effect of reduce task settings when only a fraction of resources is available (measured results).

the results with the input dataset size of 10 GB and 1 GB respectively. The graphs reflect that when less resources are available to a job (e.g., 10% of all map and reduce slots in the cluster), the offered rule of thumb setting (115 reduce tasks) could even hurt the query completion time since the expected high concurrency degree in the job execution cannot be achieved with limited resources while the overhead introduced by a higher number of reduce tasks causes a longer completion time. This negative impact is even more pronounced when the input dataset size is small as shown in Figure 7 (b). For example, when the query can only use 10% of cluster resources, the query completion time with the rule of thumb setting (115 reduce tasks) is more than 2 times higher compared to the completion time of this query with eight reduce tasks.

Analyzing Performance Trade-offs. Now, we evaluate two optimization strategies introduced in Section 3 for deriving workflow reduce task settings and analyzing the achievable performance trade-offs. Figure 8 presents the normalized resource usage under local and global optimization strategies when they are applied with different thresholds for a workflow completion time increase, i.e., *w_increase= 0%, 5%, 10%, 15%*. Figures 8 (a)-(b) show the *measured results* for two TPC-H queries with the input size of 10GB (i.e., scaling factor of 10), and Figures 8 (c)-(d) show results for two proxy queries that process 3-month data of web proxy logs. For presentation purposes, we show the normalized workflow resource usage with respect to the resource usage under the *rule of thumb* setting that sets the number of reduce tasks in the job to 90% of the available reduce slots in the cluster. In the presented results, we also eliminate the resource usage of the map stage in the first job of the workflow as its execution does not depend on reduce task settings.



(a) TPC-H Q1 (b) TPC-H Q19

(c) proxy-Q1 (d) proxy-Q2

Figure 8: Local and global optimization strategies: resource usage with different *w_increase* thresholds.

The results are quite interesting. The first group of bars in Figure 8 shows the normalized resource usage when a user aims to achieve the minimal workflow completion time (*w_increase= 0%*). Even in this case, there are 5%-30% resource savings compared to the *rule of thumb* settings. When *w_increase= 0%* the local and global optimization strategies are identical and produce the same results. However, if a user accepts 5% of the completion time increase it leads to very significant resource savings: 40%-95% across different queries shown in Figure 8. The biggest resource savings are achieved for *TPC-H Q1* and *Proxy Q1*: 95% and 85% respectively. Moreover, for these two queries global optimization strategy outperforms the local one by 20%-40%. As we can see the performance trade-offs are application dependent.

In summary, *AutoTune* offers a useful framework for a proactive analysis of achievable performance trade-offs to enable an efficient workload management in a Hadoop cluster.

5 Related Work

Several different approaches were proposed for predicting the performance of MapReduce applications [6, 5, 11, 12, 17].

In *Starfish* [6], the authors apply dynamic Java instrumentation to collect a run-time monitoring information about job execution. They create a fine granularity job profile that con-

sists of a diverse variety of metrics. This detailed job profiling enables the authors to predict the job execution under different Hadoop configuration parameters, automatically derive an optimized cluster configuration, and solve cluster sizing problem [5]. Tian and Chen [11] propose predicting a given MapReduce application performance from a set of test runs on small input datasets and a small Hadoop cluster. By executing a variety of 25-60 test runs the authors create a training set for building a model of a given application. It is an interesting approach but the model has to be built for each application and cannot be applied for parameter tuning problems. *ParaTimer* [7] offers a progress estimator for parallel queries expressed as Pig scripts [4]. In the earlier work [8], the authors design *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. These models are designed for estimating the remaining execution time of workflows and DAGs of MapReduce jobs. However, the proposed models are not applicable for optimization problems.

ARIA [12] builds an automated framework for extracting compact job profiles from the past application run(s). These job profiles form the basis of a *MapReduce analytic performance model* that computes the lower and upper bounds on the job completion time. ARIA provides an SLO-based scheduler for MapReduce jobs with timing requirements. The later work [17] enhances and extends this approach for performance modeling and optimization of Pig programs. In our work, we design a different profiling of MapReduce jobs via eight execution phases. The phases are used for estimating map/reduce tasks durations when the job configuration is modified.

MRShare [9] and *CoScan* [14] offer frameworks that merge the executions of MapReduce jobs with common data inputs in such a way that this data is only scanned once, and the workflow completion time is reduced. AQUA [15] proposes an automatic query analyzer for MapReduce workflow on relational data analysis. It tries to optimize the workflow performance by reconstructing the MapReduce DAGs to minimize the intermediate data generated during the workflow execution.

In our work, we focus on optimizing the workflow performance via tuning the number of reduce tasks of its jobs under a given Hadoop cluster configuration. We are not aware of any published work solving this problem.

6 Conclusion

Optimizing the execution efficiency of MapReduce workflows is a challenging problem that requires the user experience and expertize. In this work, we outline the design of *AutoTune* - the automated framework for tuning the reduce task settings while achieving multiple performance objectives. We observe that the performance gain for minimizing a workflow completion time has a point of diminishing return. In the future, we plan to design useful utility functions for automating the trade-off part of analysis in the optimization process.

References

[1] Apache hadoop: Best practices and anti-patterns. `http://developer.yahoo.com/blogs/hadoop/posts/2010/08/apache_hadoop_best_practices_a/`, 2010.

[2] BTrace: A Dynamic Instrumentation Tool for Java. `http://kenai.com/projects/btrace`.

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

[4] A. Gates et al. Building a High-Level Dataflow System on Top of Map-Reduce: The Pig Experience. *Proc. of VLDB Endowment*, 2(2), 2009.

[5] H. Herodotou, F. Dong, and S. Babu. No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics. In *Proc. of ACM Symposium on Cloud Computing*, 2011.

[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 5th Conf. on Innovative Data Systems Research (CIDR)*, 2011.

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

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

[9] T. Nykiel et al. MRShare: sharing across multiple queries in MapReduce. *Proc. VLDB Endow.*, 3(1-2), Sept. 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 the 8th ACM Intl. Conf. on Autonomic Computing (ICAC)*, 2011.

[13] A. Verma, L. Cherkasova, and R. H. Campbell. Resource Provisioning Framework for MapReduce Jobs with Performance Goals. *Proc. of the 12th ACM/IFIP/USENIX Middleware Conference*, 2011.

[14] 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.

[15] S. Wu, F. Li, S. Mehrotra, and B. C. Ooi. Query optimization for massively parallel data processing. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, 2011.

[16] Z. Zhang, L. Cherkasova, and B. T. Loo. Benchmarking Approach for Designing a MapReduce Performance Model. In *Proc. of the 4th ACM/SPEC Intl. Conf. on Performance Engineering (ICPE)*, 2013.

[17] Z. Zhang, L. Cherkasova, A. Verma, and B. T. Loo. Automated Profiling and Resource Management of Pig Programs for Meeting Service Level Objectives. In *Proc. of ICAC*, 2012.