Utility Scheduling for Multi-Tenant Clusters

Logan Stafman

A Dissertation

Presented to the Faculty

of Princeton University

in Candidacy for the Degree

of Doctor of Philosophy

Recommended for Acceptance

by the Department of

Computer Science

Adviser: Professor Michael J Freedman

June 2019

Abstract

The rapid increase in data size along with the complex patterns of data usage amongst data scientists presents new challenges for large-scale data analytics systems. Modern distributed computing frameworks must support complex applications that range from answering database queries to training machine learning models. As data centers have grown, managing their resources has become an increasingly important task. New applications have become popular that make traditional scheduling systems inadequate.

In this thesis, we present distributed scheduling systems aimed at increasing cluster resource utilization by taking advantage of specific characteristics of data processing applications. First, we identify a set of applications whose characteristics make them prime targets for utility-based scheduling. We then focus on two specific types of these applications in the following systems:

(i) SLAQ: a cluster scheduling system for machine learning (ML) training jobs that aims to maximize the qualities of all models trained. In exploratory model training, models can be improved more quickly by redirecting resources to jobs with the highest potential for improvement. SLAQ reduces latency and maximizes the quality of models being trained by a distributed ML cluster.

(ii) ReLAQS: a cluster scheduling system for incremental approximate query processing (AQP) systems that aims to minimize the error of all approximate results. In AQP, queries compute approximate results by sampling data. In AQP, error can be reduced more quickly by allocating resources to queries with higher error. ReLAQS reduces the latency required to reach a query result with a given level of error in a shared AQP environment.

These works demonstrate a novel set of methods that can be used in fine-grained scheduling to build responsive, efficient distributed systems. We have evaluated these systems on standard benchmark workloads and datasets, as well as popular ML algorithms, and show both reduced latency and increased accuracy of intermediary results.

Acknowledgments

I am extremely grateful to my advisor, Mike Freedman, for taking me in, supporting me, and helping me shift research directions. His patience, encouragement, and expertise have all been crucial in getting me through my PhD. The freedom he gave me to explore research topics helped me to figure out my true interests.

I also would like to thank my committee members, Kyle Jamieson, Wyatt Lloyd, Jennifer Rexford, and Amit Levy. They have all provided guidance and feedback at various stages of my time at Princeton. Wyatt has always been available to give feedback at any stage of a project. Jen has been extremely available to help with any problems I faced along the way, and for that I am so grateful.

I am thankful for the professional and personal friendships I've made with my collaborators here at Princeton. I am grateful to Harris for his contributions to SLAQ, as well as Andrew for his contributions to SLAQ and ReLAQS. Both of them have been wonderful to work with, from brainstorming to debugging.

I've enjoyed the personal friendships I've made both in and out of the Computer Science department. The SNS group members have provided technical and emotional support. Themis, Marcela, Daniel, Aaron, Robert, and David all provided invaluable help with paper drafts. In addition, I'd like to thank Sachin, Jonathan and others for their help with things from an ML perspective. I'd also like to thank my friends in the Juggling Club for helping me learn to relax: Channing, JVO, Izzy, Skye, Daniel, Justin, Jack, and so many more. Wenhao, Travis, and Sree have all provided valuable career advice and technical feedback.

My internship at VMWare provided invaluable experience, and my mentors Amitabh Banerjee and Rishi Mehta provided me with invaluable support and guidance.

This dissertation was supported by National Science Foundation Awards CNS-0953197, IIS-1250990, and NSF CNS-1763546.

Finally, I wish to thank both Rebecca and my family for always believing in me and encouraging me to complete my PhD. Their love and support have gotten me through this journey.

To my parents and sister.

Contents

List of Tables

List of Figures

Bibliographic Notes

The material presented in Chapter 3 appears in an SoCC paper (2017) co-authored with Haoyu Zhang, Andrew Or, and Michael J Freedman [125]. A more condensed version of this same work appeared at SysML (2017), also co-authored by Haoyu Zhang, Andrew Or, and Michael J Freedman [126]. Most of the material presented in Chapter 4 is currently under submission. This work is co-authored by Andrew Or and Michael J Freedman.

Chapter 1

Introduction

1.1 Distributed Data-Driven Approximate Applications

Modern data scientists use distributed computing platforms to process huge amounts of data. They write applications to analyze data, discover relationships, and make decisions about real-world problems. The applications using these platforms vary greatly – while some simply apply a single operation to each piece of data, others are more complex. The complex nature of these applications, combined with the spectacular growth of data presents new challenges to the distributed systems these applications are built upon.

Limited Cluster Resources The growth in resource demands that naturally accompanies these applications has outpaced the growth in processor capabilities, as Moore's Law comes to an end [107]. In the past, Moore's Law guaranteed that performance gains would come easily with newer hardware. However, as distributed data analytics jobs grow, the cost of extra resources has lead to high resource contention. Distributed data analytics jobs must therefore be resource-aware to ensure reasonable job latencies.

Users Share Clusters The users of these distributed analytics clusters tend to share these clusters in an effort to save money. They share them with other data scientists, other organizations, or even other jobs they themselves submit. When data scientists submit jobs

to these clusters, the usage patterns follow their diurnal working patterns. When jobs are instead submitted by applications, they tend to follow the usage patterns of the users of the applications. When many users or organizations share the same cluster, they can expect to experience contention as many jobs are submitted simultaneously.

Approximate Applications are Popular Within the data analytics world, machine learning applications have become wildly popular. The vast majority of these are approximate in nature – the underlying optimization problems they solve do not have an exact answer, and instead iterate towards a more accurate result. On the other hand, the huge jump in data sizes has lead to the adoption of distributed approximate applications that give slightly erroneous answers with significantly shorter latencies than traditional applications. These popular applications provide unique challenges, exploring a tradeoff space that provides opportunities for reducing some of the contention faced by these clusters.

1.2 Challenges for Existing Distributed Data-Driven Approximate Systems

Modern distributed systems face several key challenges as the machines they run on and the types of applications submitted to them change. Furthermore, as they become more accessible, users may have less knowledge of the system, moving some key design decisions from the shoulders of application developers to the system itself.

Distributed Data Analytics are Complex The applications that analyze large amounts of data on shared clusters are often complex computational applications. In Apache Spark, jobs can be composed of hundreds of computational stages [121], greatly affecting the parallelizability of the application and causing computational slowdown due to the straggler effect [91]. To further complicate these jobs, users may use libraries that hide the complex nature of these applications, such as SparkSQL, causing the applications to be more

complex than expertly-tuned applications. As web dashboards and other data-heavy web applications have become more common, we've seen the rise of complex queries in high volume.

In ML workloads, jobs can take hundreds or thousands of steps to converge to an answer. Moreover, in distributed clusters, decisions about how to synchronize ML models complicate things, introducing tradeoffs between computational and network bandwidth overheads. As ML has progressed in recent years, the hyperparameter space has exploded, leading practitioners to explore thousands of combinations of settings in order to find the most highly tuned models [82]. This further taxes the resources available as single users may be submitting many jobs to quickly obtain quality results.

The Growth of Data It is well known that large datasets are growing much more quickly than processing power [39]. In Machine Learning, training datasets have grown huge as technology companies collect huge amounts of information about users. This is born out by the MovieLens dataset with over 22 million ratings [59] or ImageNet [49] with 14 million images precategorized. As deep learning has become more popular, these data sizes have only grown. These large datasets introduce difficulties as ML requires scanning the entire dataset many times. Furthermore, growing model sizes introduce interesting and complicated tradeoffs.

In distributed databases, the growth of data introduces challenges as well. Query plans must include resource availability and other information about a cluster's nodes to get the most out of a cluster [84].

1.3 Resource Management in Distributed Clusters

As clusters are shared between a group of users and organizations, understanding how to equitably partition resources between users is imperative. Resource management systems have been adopted to coordinate all of the resources a cluster shares. These resource sched-

ulers are sometimes themselves distributed, to avoid becoming a bottleneck [92] and to be more robust to failures.

Typically, the scheduler can be organized in a two-layer architecture. The *job-level scheduler* is in charge of allocating resources evenly between different jobs active in the system, representing different computing tasks belonging to different users. The policies of this layer generally aims to partition the resources of the underlying system equally between jobs, attempting to obtain *fair-share* of resources.

By contrast, the *task-level scheduler* is in charge of assigning tasks, or subcomputations of jobs, to individual nodes to maximize the usage of the resources available in the cluster. The policies this layer uses to accomplish this typically involve intelligently partitioning jobs to achieve an even balance between scheduling overhead and responsiveness. For example, a task belonging to a SQL processing job may complete a filter on a single tuple; this would incur significant overhead as each individual tuple must be scheduled on a node in the system. On the other hand, a task representing the same filter on 1 GB worth of tuples would have high latency, leading to bottlenecks in processing.

Traditional cluster resource management systems treat these applications as blackboxes – while they may get general information from the user about job deadlines, priorities, or other strategies, the applications themselves provide no data to the scheduler.

In this thesis, we present the design of a scheduler for a subset of big data analytics approximate applications. These applications are popular in large companies and can be run on thousands of nodes. Thus, increasing the utility of the work done on these clusters can both help these systems monetarily as well as from an energy consumption standpoint.

- We identify the properties of approximate applications for which our approach is best suited, and discuss some specific examples of these applications. We describe the potential benefits of scheduling these approximate applications by their utilities compared to standard solutions. We also discuss the requirements these applications

must meet, the challenges we will need to tackle to take our approach, and the usage patterns these applications typically are used in that further motivate our approach.

- Machine Learning is a specific example of these applications and we delve in to the unique challenges the most popular ML applications face. We designed and evaluated SLAQ, a scheduling system for ML applications.
- Online Aggregation is another example of these applications. We design ReLAQS to schedule these applications, and discuss some of the unique challenges facing Online Aggregation, as well as the common settings in which it is used.

1.4 Contributions

SLAQ is a cluster scheduler for Machine Learning jobs that aims to maximize the quality of the models. SLAQ allocates resources based on an ML job's model quality and the overall system contention. The key intuition behind SLAQ is that the usefulness of computation in ML training jobs is reduced over time, producing diminishing returns. Instead of wasting computation on these jobs, we argue that more resources should be given to other jobs whose potential for progress is higher. By continually monitoring the history of model quality and runtime, SLAQ generates quality predictions for future iterations. SLAQ estimates the impact resource allocation would have on model quality and adjusts the resource allocations of all active jobs to best utilize the limited cluster resources. The SLAQ scheduler is lightweight and fine-grained to allow resource allocations to quickly adjust to changes in job progress.

ReLAQS is a cluster scheduler for Online Aggregation jobs that aims to minimize the error of the intermediate results. ReLAQS allocates resources based on an Online Aggregation job's estimated error and the overall system contention. As with SLAQ, ReLAQS estimates the usefulness of computation in online aggregation jobs as they complete. We show that online aggregation jobs produce diminishing returns after the initial mini-batches have completed. ReLAQS monitors the results produced by each mini-batch, and uses that

to estimate the total progress of each online aggregation job. Using that information, ReLAQS redistributes resources to minimize the total error of all approximate results of active jobs in the cluster.

Chapter 2

The case for utility-based scheduling for approximate applications

2.1 Introduction

As outlined in Chapter 1, high resource contention in shared clusters can lead to high latency. As a many jobs are submitted simultaneously to a shared cluster, the cluster can become bogged down. In this chapter, we will discuss a set of applications that can benefit from utility-based scheduling. We will lay out how to identify these applications, the properties these applications must have, the potential benefits of utility-based scheduling, and the potential drawbacks.

2.2 Approximate Applications and Their Properties

An approximate application is one that returns an approximate result in exchange for a shorter runtime, or fewer resources. Some of these applications behave similarly to an exact application, returning one approximate result after completing computation. Incremental approximate applications, on the other hand, return approximate results throughout the execution of an application. Below, we've shown three examples of an application that finds the average of numbers 1 to 100. Algorithm 1 represents an exact application that runs to

completion before returning an exact answer. Algorithm 2 is an approximate version of the same application that returns an approximate result after only a fraction of the time. Lastly, Algorithm 3 is the incremental approximate version of that same application, returning approximate results throughout the course of the run, increasing in accuracy over time. These incremental approximate applications are what we will be focusing on in this chapter.

Algorithm 1 Exact

```
1: numbers ← scramble(1,100)
2: sum ← 0
3: for i ∈ [1,...,100] do
4:     sum ← sum + numbers[i]
5: print(sum/100)
```

Algorithm 2 Approximate

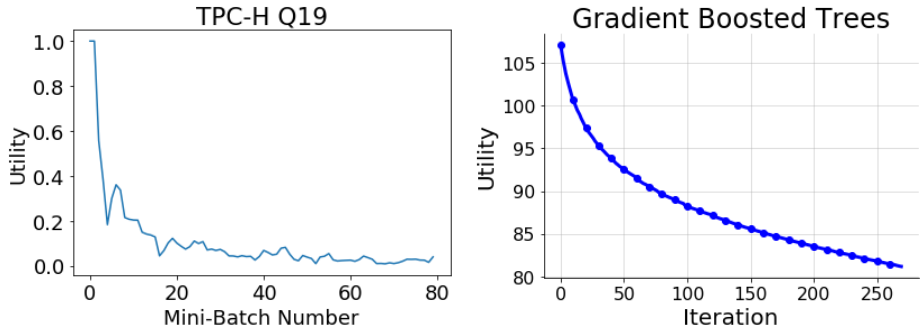
```
1: numbers ← sample(scramble(1,100),10)
2: sum ← 0
3: for i ∈ [1,...,10] do
4:     sum ← sum + numbers[i]
5: print(sum/10)
```

Algorithm 3 Incremental Approximate

```
1: numbers ← scramble(1,100)
2: sum ← 0
3: for i ∈ [1,...,100] do
4:     sum ← sum + numbers[i]
5:     if i mod 10 == 0 then
6:         print(sum/i)
```

Goodput versus Throughput In computer networks, the concept of throughput describes how many bits are transferred from one application to another over a given amount of time. By contrast, goodput describes how many *useful* bits are transferred from one application to another. If an application sends the same information over and over, its throughput exceeds its goodput. In a resource-constrained environment in which overall

bandwidth is limited, it would make sense to throttle applications with a low goodput to throughput ratio, allowing the network to transmit more useful data. Similarly, in a shared cluster, if an application produces less goodput, or *utility* with the same resources as another application, it would make more sense to provide those resources to the application that can produce more useful computation. In fact, many schedulers already do this – a cluster manager may declare a higher priority for one application over another, and popular schedulers will provide more resources to those applications, increasing the overall utility of the whole system.



(a) TPC-H Q19 Estimated Utility (b) Gradient Boosted Trees Utility

Figure 2.1: These applications are examples of applications whose utility fades over the runtime of an application. Note that utility drops sublinearly depending on properties unique to each application.

Diminishing Returns in Distributed Data Analytics When applications have constant utility over the entire course of the runtime, it's somewhat trivial to apportion resources accordingly. However, in many distributed data analytics applications, the usefulness of the computation is not constant over the runtime of the application. Many incremental approximate applications, including machine learning applications, optimization problems, and online aggregation produce diminishing returns over time. In particular, these applications must produce results that have value at intermediate results for users. In Figure 2.1, we show a few sample applications and their estimated results' utility over time. While each application uses the same amount of resources throughout its full run, the utility of

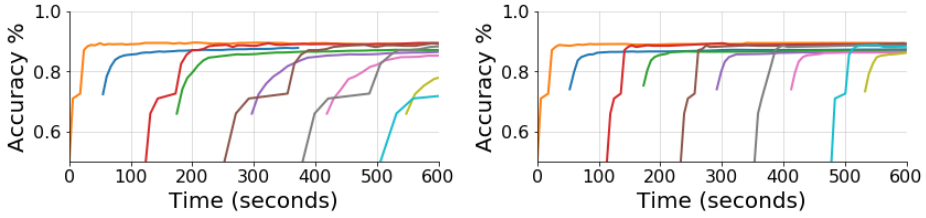
their intermediate results drops over the course of their run. Note that while in one of these applications (Gradient Boosted Trees), the rate at which utility drops over time is quite smooth and predictable, in the other (Online Aggregation on the TPC-H benchmark), utility does not necessarily change in a clear way. This will affect some of our decisions when we actually schedule these applications.

Real-World Distributed Data Analytics Use Cases Understanding how data analysts use clusters is an important step for designing a scheduler for these applications, and helping to define the goals our system should aim for. Distributed data analytics system users can have a wide range of expertise in the underlying distributed system. For that reason, it is imperative that our scheduler asks users to modify their applications as minimally as possible, as well as asking for little domain knowledge. Furthermore, users have varying needs from approximate applications; some desire precise answers with high latency, while others desire more erroneous answers with low latency. Others still may not know how much accuracy they require until they've already seen some approximate results. This complicated and varying set of use cases creates a unique challenge in scheduling approximate data analytics workloads.

2.3 Scheduling Approximate Applications

In order to take advantage of these applications whose utilities change over the course of their run, we need to first understand what the potential benefits are of scheduling based on utility, as well as the key challenges that we have to tackle.

Benefits of Utility Based Scheduling By using utility to schedule applications, we can expect to reduce latency and improve the overall accuracy of results of applications submitted to our system. Because these applications produce diminishing returns, we avoid wasting resources on jobs producing only minor improvements. For example, in Figure



(a) Resource Fair Scheduling (b) Utility Based Scheduling

Figure 2.2: Utility based scheduling allows applications to reach higher accuracy solutions much more quickly.

2.2, we show a sequence of applications joining a cluster every 60 seconds. These applications produce approximate results and their accuracy increases over time. In 2.2(a), as more applications join the cluster, each application takes longer to reach high accuracy, as each is given a smaller and smaller proportion of the resources. By contrast, in 2.2(b) those same applications reach high accuracy almost as quickly even as the cluster becomes more crowded.

It is important to note that scheduling is a zero-sum game – necessarily, giving one application more resources removes those resources from another, in a cluster where all resources are being used. This means that applications that are nearly completed and produce small amounts of utility may have high latency to reach their small benefits. This is a tradeoff we further discuss in Chapter 4. We believe that the latency and accuracy benefits a scheduler based on utility could provide to many applications are worth the penalty of high latency for those applications that already have highly accurate results.

This tradeoff is one of the key challenges of creating a scheduler based on utility. We also have several other challenges that we will need to tackle.

Identifying Utility First and foremost, understanding how much utility a particular application provides a user at any given point during its execution is crucial to creating a scheduler that can maximize the overall utility of all jobs in a system. Not only do different applications produce different types of utility, but we also must take into account the inten-

tions of the user. For example, if a user is *only* interested in exact results, all intermediate approximate results provide zero utility to the user. However, the users of many distributed data analytics are interested in intermediate results.

Moreover, we must be able to find a good metric which not only understands an application's utility, but also is comparable between applications. This can be challenging to do online, as determining an incremental result's accuracy may not be known until the end of an application. We discuss several ways of tackling this problem for each application.

Predicting Utility Schedulers typically schedule at every epoch – every few seconds, they redistribute resources based on any new information they've received during the last epoch. In order to appropriately redistribute resources, we need to be able to estimate, for each application, how much utility that application will be able to create with those resources over the next epoch. This affects design decisions twofold: The way we measure utility must be chosen to be somewhat predictable, and we have to decide the best way to predict this in a lightweight manner.

Redistributing Resources Another key challenge we faced was how to redistribute resources between applications. As many schedulers provide a mechanism to prioritize applications, a simple way to reapportion resources is to use each application's potential utility as a prioritization weight. While this does provide modest improvements, it is only an approximation. Instead, we approach this challenge by optimizing the total utility of all applications, taking into account each application's parallelizability and predicted future improvement.

Low Impact to Users Lastly, it was important to us that we be able to make these changes with as little change to users' applications as possible. By introducing almost no work to application developers, they can easily use any scheduler we make without necessarily

understanding the scheduler, or even the ML or online aggregation applications they themselves are running.

2.4 Related Work

Approximate Applications In this thesis, we discuss online aggregation [62] and machine learning. Both are examples of incremental approximate applications, and have deep bodies of work covering them, including some work focused on reducing resource consumption in both types of applications. However, the approaches taken in these works are typically orthogonal to our approach and generally would work in tandem with our approach. In Machine Learning, parameter servers and work on reducing synchronization overheads have aimed to reduce resource consumption [86, 15, 5, 78, 37, 102, 14]. In online aggregation, sampling techniques to reduce overhead and deduplication of computation have made similar attempts to reduce resource usage [115, 17].

Priority Scheduling Utility has often been used in the past by schedulers. In modern clusters, most have a concept of *priority* by which one user or organization may be deemed more important or mission critical and jobs submitted by that user will be given more resources, effectively increasing the overall utility of the cluster's output, from the perspective of the cluster manager. One example of priority-based scheduling is in OS-level schedulers. UI-bound threads may be given higher priority to avoid disruption to the user experience.

Cluster Scheduling Systems A lot of work has been aimed at partitioning resources evenly between applications in clusters. Existing cluster schedulers [119, 63, 52, 27, 57, 64] primarily focus on resource fairness, job priorities, cluster utilization, or resource reservations, but do not take job progress rates into consideration. These systems provide not only a baseline against which we can benchmark our solutions, but also a set of techniques and hierarchical architecture we can use as the baseline for our schedulers. However, none

of this work considers the possibility that a single application could have different utilities to a user at different points during its execution.

Chapter 3

SLAQ: Scheduling Machine Learning Applications for Quality

Machine learning (ML) is an increasingly important tool for large-scale data analytics, including online search, marketing, healthcare, and information security. A key challenge in analyzing massive amounts of data with ML arises from the fact that model complexity and data volume is growing much faster than hardware speed improvements. Thus, time-sensitive machine learning on large datasets necessitates the use and efficient management of cluster resources. Three key features of ML are particularly relevant to resource management.

ML algorithms are intrinsically approximate. ML algorithms generally consist of two stages: *training* and *inference*. The training stage builds a model from a training dataset (e.g., images with labeled objects), and the inference stage uses the model to make predictions on new inputs (e.g., recognizing objects in a photo). ML models are intrinsically approximate functions for input-output mapping. We use *quality* to measure how well the model maps input to the correct output.

ML training is typically iterative with diminishing returns. While the inference stage is often lightweight and can run in real-time, the training stage is computationally expensive

and usually requires multiple passes over large datasets. It generates a low-quality model at the beginning and improves the model's quality through a sequence of training iterations until it converges. In general, the quality improvement diminishes as more iterations are completed.

Training ML is an exploratory process. ML practitioners retrain their models repeatedly to explore feature validity [22], tune hyperparameters [103, 83], and adjust model structures [58] before they operationalize their final model, which is deployed for performing inference on individual inputs. The goal of retraining is to get the final model with the best quality. Since ML training jobs are expensive, practitioners in experimental environments often prefer to work with more approximate models trained within a short period of time for preliminary validation and testing, rather than wait a significant amount of time for a better trained model with poorly tuned configurations. In fact, algorithm tuning is an empirical process of trial and error that can take significant effort, both human and machine. With the exponential growth of data volume, the cost of decision making on model configurations will likely continue to increase.

Many ML frameworks have been developed [86, 8, 15, 5] to run large-scale training jobs in clusters with shared resources. Existing schedulers primarily focus on *resource fairness* [119, 63, 52, 27, 57, 64], but are agnostic to model quality and job runtime. During a burst of job submissions, equal resources will be allocated to jobs that are in their early stages and could benefit significantly from extra resources as those that have nearly converged and cannot improve much further. This is not efficient.

We present SLAQ, a cluster scheduling system for ML training jobs that aims to maximize the overall job quality. SLAQ dynamically allocates resources based on job resource demands, intermediate model quality, and the system's workload. The intuition behind SLAQ is that in the context of approximate ML training, more resources should be allocated to jobs that have the most potential for quality improvement.

SLAQ leverages the fact that most ML training algorithms are implemented as an iterative optimization process. By continually monitoring the history of quality improvement and runtime, SLAQ generates highly-tailored and accurate quality predictions for future training iterations. SLAQ estimates the impact of resource allocation on model quality, and explores the quality-runtime trade-offs across multiple jobs. Based on this information, SLAQ adjusts their resource allocations of all running jobs to best utilize the limited cluster resources. The SLAQ scheduler is designed to be dynamic and fine-grained, so that resource allocations can adapt quickly to jobs' quality and the system's workload changes.

Challenges and solutions. In designing SLAQ, we had to overcome several technical challenges.

First, ML training algorithms measure the quality of models with tens of different metrics, which makes it difficult to compare the training progress of different jobs. SLAQ normalizes these metrics using the reduction of loss values. These intermediate quality measures are reported directly by the application APIs. Our normalization effectively unifies the quality measures for a broad set of ML algorithms.

Second, SLAQ should be able to precisely predict the impact that an extra unit of resources would have on the quality and runtime of ML training jobs. Previous work [110] predicts a job's runtime based on its computation and communication structure, but it requires that the job be analyzed or profiled offline. Unfortunately, the significant overhead of this offline analysis is prohibitive for our exploratory setting. SLAQ uses online prediction: it predicts the time and quality of the coming iterations based on statistics collected from previous iterations.

SLAQ supports configurable high-level goals when scheduling jobs. When maximizing the aggregate quality improvement, it can best utilize the cluster resources and achieve a higher total quality gain across all jobs. When maximizing the minimum quality, SLAQ

can achieve the equivalent of max-min fairness applied to quality (rather than resource allocation).

While SLAQ works with a large class of important ML algorithms, some non-convex ML algorithms are not currently supported. The convergence properties and optimization of these algorithms are being actively studied, and we leave scheduling support for these algorithms to future work.

We implemented SLAQ as a new scheduler within the Apache Spark framework [120]. SLAQ can use its quality-driven scheduling for many of the ML algorithms available in MLlib [86], Spark's machine learning package. In fact, SLAQ supports unmodified ML applications using existing MLlib optimizers, as well as applications using new optimization algorithms with only minor modifications. We evaluate various distinct ML training algorithms on datasets collected from various online sources. We found that SLAQ improves the average quality by up to 73% and reduces the average delay by up to 44% compared to fair resource scheduling.

3.1 Background and Motivation

The past several years have seen a rapid increase in both the volume of data used to train ML models and the size and complexity of these models. Growth in the performance of the underlying hardware, however, has not caught up, thus placing higher demands on the computational resources used for this purpose.

An important way that data scientists cope with these demands is to leverage more approximate models for preliminary testing, in order to exclude bad trials and iterate to the right configuration. A significant amount of time and resource usage can potentially be saved because of the iterative nature of ML optimization algorithms, and the diminishing returns of quality improvements during the training iterations. Today's schedulers, however, do not provide a ready means to follow this strategy; a traditional max-min fair

scheduler (similarly, the dominant resource fair scheduler [52]) ensures fair *resource* allocation without considering the potential of these resources to improve model quality.

This section motivates and provides background for SLAQ. §3.1.1 describes the iterative nature of the ML training process and how it is characterized by diminished returns. We introduce the exploratory training process in §3.1.2 and describe current practices in §3.1.3. We discuss the problems with existing cluster schedulers and propose our quality-aware scheduler in §3.1.4.

3.1.1 ML Training: Iterative Optimization Process

The algorithms used for the ML training process typically include a dataset specification, a loss function, an optimization procedure, and a model [54]. A machine learning model is a parametric transformation $f_\theta : X \mapsto Y$ that maps input variables to output variables, and it typically contains a set of parameters θ which will be regularly adjusted during the training process. The loss function represents how well the model maps training examples to correct output, and is often combined with a regularization term to incorporate generalizability. Training machine learning models can be summarized as optimizing the model parameters to minimize the loss function when applying the model on a dataset.

When the machine learning model is nonlinear, most loss functions can no longer be optimized in closed form. Algorithms such as Gradient Descent, L-BFGS and Expectation Maximization (EM) are widely used in practice to *iteratively* solve the numerical optimization of the loss function. As the sizes of the dataset and model grow, the batch algorithms can no longer solve the optimization problem efficiently. Instead, various new algorithms have been proposed to improve the efficiency of the optimization process in an iterative and distributed fashion. For example, stochastic gradient descent (SGD) [29] reduces computationally complexity by evaluating the loss function and gradient on a randomly drawn subset of the overall dataset in each iteration.

The training process with the iterative optimization algorithms can be viewed as a refinement loop of the model. After initializing the parameter values (e.g., with random

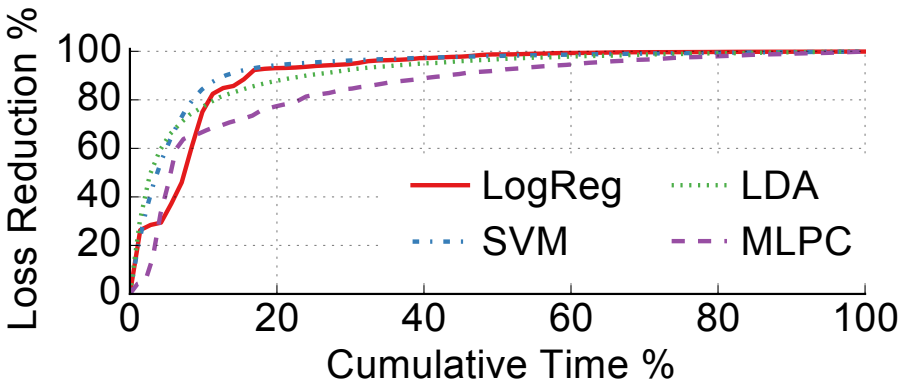


Figure 3.1: Cumulative time to achieve different percentages of loss reduction with four jobs: Logistic Regression (LogReg), Support Vector Machine (SVM), Latent Dirichlet Allocation (LDA) and Multi-Layer Perceptron Classifier (MLPC). Job convergence is defined to be $1/10000$ of initial loss reduction.

values), the optimization algorithms calculate changes on parameters in order to reduce the loss function, and update the model with new parameter values. This process continues until the decrease in the loss function falls below a certain threshold, or until a preset number of iterations have elapsed.

Another approach that some ML algorithms take is ensemble learning. Instead of training a complicated model with a large number of parameters, these algorithms focus on aggregating results from multiple diverse but small *submodels*. For example, boosting algorithms improve the accuracy of the model classifier by examining the errors in results, adding new submodels to the ensemble, and adjusting the weights of the set of submodels. Boost aggregating (bagging) algorithms train multiple submodels on different subsets of the training data by sampling with replacement. The training process of the ensemble models involves both iteratively refining each submodel, and iteratively adding new submodels or adjusting the weights of existing components.

When training a machine learning model, the first several iterations generally boost the quality very quickly. This is because the initial parameters of a model are generally set randomly. However, for most ML training algorithms, the quality improvements are subject to diminishing returns; iterations in later stages continue to cost the same amount of computa-

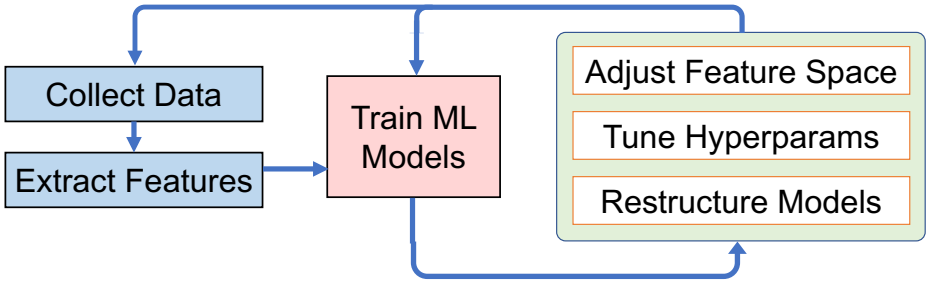


Figure 3.2: Retrain machine learning models.

tional resources while making only marginal improvements on model quality as the results finally converge. For example, error in gradient descent algorithms on convex optimization problems often converges approximately as a geometric series [33]. Theoretically, at the kth iteration, the loss function reduction is $O(\mu^k)$, where μ is the convergence rate ($|\mu| < 1$). In general, loss reduction (quality improvement) diminishes as more iterations are completed.

Figure 3.1 plots the relative cumulative time to achieve different percentages of loss reduction. For example, it takes 20% time for the SVM job to reduce loss by 95%, and 80% time to further reduce it until convergence. Jobs for ML algorithm debugging and model tuning only require the training process to be almost completed to tell potentially good configurations from bad trials, and thus could save a lot of time and resources.

The law of diminishing returns applies in many other data analytics systems in addition to machine learning. Sampling-based approximate query processing systems compute approximate results by processing only a sample of the entire dataset in order to reduce resource usage and processing delay [17, 25, 21, 109]. Databases can also take advantage of online aggregation to incrementally refine the approximated results of SQL aggregate queries [123, 62, 93]. Using the *error* or *uncertainty* as a measurement of quality in these queries, we can observe that in most cases the convergence rate of these metrics are also monotonically decreasing.

3.1.2 Retraining Machine Learning Models

Training machine learning models is not a one-time effort. ML practitioners often train a model on the same dataset multiple times for exploratory purposes. This process provides early feedback to practitioners and helps direct their search for high quality models.

Feature engineering. Many ML algorithms require a featurized representation of the input data for efficient training and inference. For example, a speech recognition algorithm utilizes the discretized frequency features extracted from continuous sound signals with Fourier transforms and knowledge about the human ear [108]. Identifying exactly the useful features that yield the best quality relies on both domain knowledge and many training experiments.

Hyperparameter tuning. Many ML models expose *hyperparameters* that describe the high-level complexity or capacity of the models. Optimal values of these hyperparameters typically cannot be learned from the training data. Examples of hyperparameters include the number of hidden layers in a neural network, the number of clusters in a clustering algorithm, and the learning rate of mode parameters. It is desirable to explore different combinations of hyperparameter values, train multiple models, and use the one that gives the best result.

Model structure optimization. To ship ML models and run inference tasks on mobile and IoT devices, large models need to be compressed to reduce the energy consumption and accelerate the computation. Various model compression techniques have been developed [45, 58]. These methods usually prune the unnecessary parameters of the model, retrain the model with the modified structure, and then prune again. This requires training the same job multiple times to get the best compression without compromising the quality of the model.

In addition, the interactions between features, hyperparameters and model structures make it even harder to search for the best model configuration. For example, features are often correlated with one another, and modifying the set of features also requires recalibrating the hyperparameters (such as learning rate). Expensive model configuration decisions demand highly efficient resource management in shared clusters.

3.1.3 Current Practices in ML Training

When exploring the ML model configuration space, users often submit training jobs with either a time cutoff or a loss value cutoff. Both monitoring heuristics are widely used in practice but have significant drawbacks.

Training ML models within a fixed time frame often results in unpredictable quality. This is because it is often difficult to predict a priori what the loss values will be at the deadline. More importantly, when a training job shares cluster resources with other jobs, the number of iterations completed by the deadline also depends on the cluster's workload and the scheduler's decisions.

A fixed loss (or fixed Δloss) cutoff is also difficult to reason about. Loss values in different algorithms are different in magnitude and have completely different meanings (further explained in §3.3.1). Additionally, with more complicated model structures and training algorithms, it is not rare to see the convergence rate of loss function fluctuate due to stochastic methods and model staleness [38]. Fixed loss values also make users lose the potential to gain further improvement on the training.

Some users choose to manually monitor the loss function values during the training process and stop the job when they think the models are good enough. However, large-scale ML jobs could take hours or even days to complete, which makes the monitoring impractical.

In the context of exploratory ML training, it is desirable to explore the quality-runtime trade-off across multiple concurrent jobs. SLAQ automates this process and obviates the need for the user to reason about arbitrary trade-offs. SLAQ flexibly fulfills a broad range of

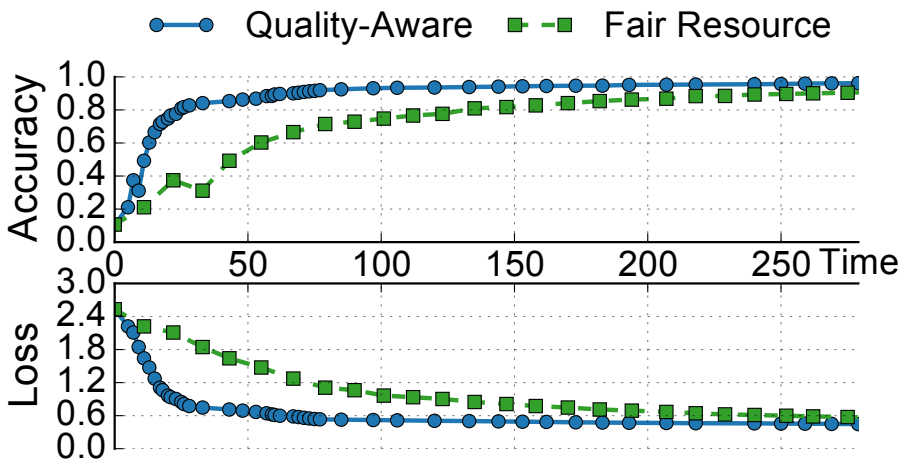


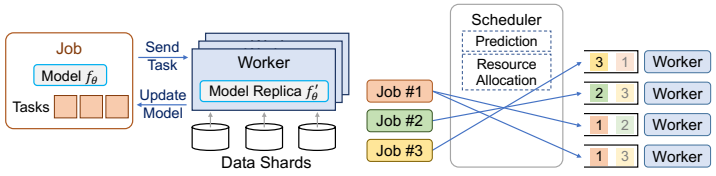
Figure 3.3: Accuracy (top) and loss function values (bottom) of a job with resources allocated by a quality-aware scheduler and a fair scheduler. Accuracy (percentage of correctly predicted data points) is evaluated on a testing dataset at the end of each training iteration. The more resources allocated to a job, the faster an iteration can be finished.

requirements for quality and delay of ML trainings, from approximate but timely models, to more traditional accurate model training. It allows users to stop jobs early before perfect convergence, and obtain a model with a loss function converged enough with much shorter latency.

3.1.4 Cluster Scheduling Systems

A cluster scheduler is responsible for managing resource allocation across multiple jobs. Modern data analytics frameworks (such as Hadoop [2], Spark [120], etc.) typically have two layers of scheduling: the job-level scheduler allocates resources to concurrent jobs running on the workers, while the task-level scheduler focuses on assigning tasks within a job to the available workers.

Existing job-level schedulers (Yarn [119], Mesos [63], Apollo [32], Hadoop Capacity [57], Quincy [64], etc.) mostly allocate resources based on resource fairness or priorities. For ML training jobs, however, these schedulers often make suboptimal scheduling decisions because they are agnostic to the progress (quality improvement) within each job. We argue that the scheduler should collect quality and delay information from each job and



(a) Distributed ML Training (b) Scheduler Architecture

Figure 3.4: Running ML training jobs with SLAQ.

dynamically adjust the resource allocation to optimize for cluster-wide quality improvement.

SLAQ is a *fine-grained* job-level scheduler: it focuses on the allocation of cluster resources between competing ML *jobs*, but does so over *short time intervals* (i.e., hundreds of milliseconds to a few seconds). Scheduling on short intervals ensures the continued rebalancing of resources across jobs, whose iteration time varies from tens to hundreds of milliseconds.

In a shared cluster with multiple users constantly submitting their training jobs, Figure 3.3 shows how the accuracy and loss values of one job change over time. With the fair scheduler, the job receives its fair share of cluster resources throughout its lifetime. A key observation here is that if we had given this job more resources in its early stages, its accuracy (loss) could have increased (decreased) much faster. SLAQ does exactly this, allocating more resources to the job when its potential improvement is large. In particular, the job was able to achieve 90% accuracy within a much shorter time frame (70s) with SLAQ than with the fair scheduler (230s). Especially for exploratory training jobs, this level of accuracy is frequently sufficient.

3.2 System Overview

SLAQ is a cluster management framework that hosts multi-tenant approximate ML training jobs running on shared resources. A centralized SLAQ scheduler coordinates the resource allocation of multiple ML training jobs. As shown in Figure 3.4(a), each job is composed

of a set of tasks. Each task processes data based on the ML algorithm on a small partition of the dataset, and can be scheduled to run on any node. The driver program contains the iterative training logic, generates tasks for each iteration, and tracks the overall progress of the job. In the case of training ML models, a task generates an update to the model parameters based on a partition of the training dataset. The duration of a task typically ranges from tens of milliseconds to a few seconds. When the tasks finish processing the data, the updates from all tasks are aggregated and sent back to the job driver program to update the primary copy of the model.

Similar to many cluster management systems, SLAQ divides machines into smaller *workers*, which is the minimum unit of resource to run a task. Figure 3.4(b) shows that each job driver, at a certain time, can send tasks to the workers allocated to that job in the cluster.

The SLAQ scheduler directly communicates with the drivers of currently running jobs to track their progress and update their resource allocation periodically. At the beginning of each scheduling epoch, SLAQ allocates resources between all the jobs based on system workload, the demands, and progress of the jobs. The scheduler reclaims workers back from some job drivers, and reallocates them to other jobs for better system-wide performance goals. Note that this is very different from many of the existing cluster managers [119, 57] which only statically allocate resources to jobs before they get started.

We made this decision due to two reasons. First, unlike general batch processing, jobs that train ML models are typically iterative and usually need longer time to complete. Scheduling only at the start of the job is too coarse-grained and can easily lead to starvation or underutilization of system resources. Second, the quality improvement of the training jobs often changes rapidly (as described in §3.1.1). Fixed allocation makes the scheduler unable to adapt to jobs' changes in quality improvement and resource demands.

Algorithm	Acronym	Type	Optimization Algorithm	Dataset
K-Means	K-Means	Clustering	Lloyd Algorithm	Synthetic
Logistic Regression	LogReg	Classification	Gradient Descent	Epsilon [13]
Support Vector Machine	SVM	Classification	Gradient Descent	Epsilon
SVM (polynomial kernel)	SVMPoly	Classification	Gradient Descent	MNIST [11]
Gradient Boosted Tree	GBT	Classification	Gradient Boosting	Epsilon
GBT Regression	GBTReg	Regression	Gradient Boosting	YearPredictionMSD [10]
Multi-Layer Perceptron Classifier	MLPC	Classification	L-BFGS	Epsilon
Latent Dirichlet Allocation	LDA	Clustering	EM / Online Algorithm	Associated Press Corpus [4]
Linear Regression	LinReg	Regression	L-BFGS	YearPredictionMSD

Table 3.1: Summary of ML algorithms, types, and the optimizers and datasets we used for testing.

3.3 Design

This section describes the mechanisms by which SLAQ addresses its key challenges. First, how to normalize quality measures between distinct jobs in order to determine how quickly they are increasing (or not) in quality relative to one another (§3.3.1). Second, how SLAQ uses jobs' resource usage and quality information to precisely predict the impact of resource allocation in an online fashion (§3.3.2). Third, how SLAQ allocates resources to maximize system-wide quality improvement (§3.3.3).

3.3.1 Normalizing Quality Metrics

As explained in §3.1.1, ML training algorithms are designed to be an optimization process which iteratively minimizes a loss function, and thus improves the model's quality. ML algorithms use various different measurement metrics to indicate the quality of model training. Though comparing a single job's quality improvement across iterations is simple, comparing these metrics across different jobs presents a challenge. To schedule for better overall quality, we need to compare the quality metrics across different jobs. This enables SLAQ to trade off resources and quality between jobs.

One straightforward solution is to use a universal metric such as *accuracy* to measure the model quality. Accuracy represents the percentage of correctly predicted data points, and the range is always from 0 to 1. Similarly, the F1 score, ROC curve, and confusion

matrix also measure the model quality taking the false positive and false negative ratios and multi-class results into consideration [97]. While these metrics are intuitively understandable for classification algorithms, they are not applicable to non-classification algorithms such as regression or unsupervised learning. In addition, accuracy and similar metrics require constructing a model and evaluating that model against a labeled validation set, which introduces an additional overhead to the job.[1]

Loss normalization. In contrast to the accuracy metrics, the loss function is calculated by the algorithm itself in each iteration, incurring no additional overhead. However, each algorithm's loss function has a different real-world interpretation. The range, convexity, and monotonicity of the loss functions depend on both the models and the optimization algorithms [54]. Directly normalizing loss values requires a priori knowledge of the loss range, which is impractical in an online setting.

For example, clustering algorithms (e.g., K-Means) use the *sum of squared distances to the cluster centroids* as the loss function. Classification and regression algorithms (e.g., SVM, Linear Regression, etc.) commonly use hinge or logistic gradient loss which represents *discrepancy of prediction* on the training data. The range of the measured values can vary by orders of magnitude: K-Means on our synthetic dataset reduces the loss from 300 down to 0, and the range highly depends on the absolute coordinates of the data points; on the other hand, SVM on a handwritten digit recognition dataset [11] reduces the loss from 1 down to 0.4. Unfortunately, there are no known analytical models to predict these ranges without actually running the training jobs.

Based on the convergence properties of loss functions (further explained in §3.3.2), we choose to normalize the *change* in loss values between iterations, as opposed to the loss values themselves. Most optimizers used in training algorithms try to reduce the values of loss functions, and for convex optimization problems, the values decrease monotoni-

[1] Validation is commonly used in ML training to prevent overfitting. Due to the overhead, however, model evaluation on the validation set is usually performed once every several iterations, not every iteration.

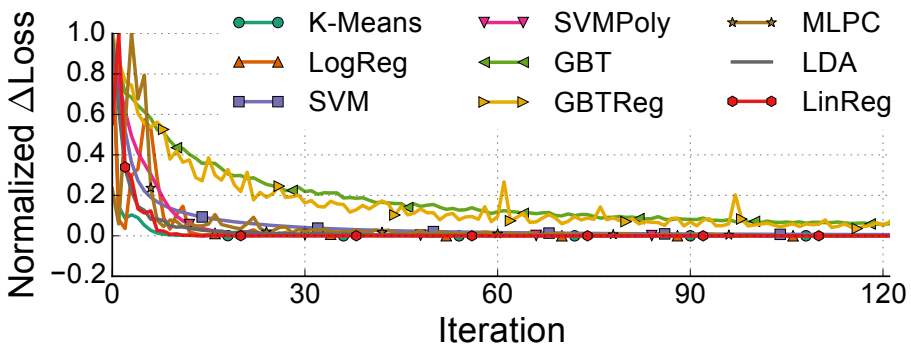


Figure 3.5: Normalized ΔLoss for ML algorithms.

cally [33]. The convergence rate, because of the diminishing returns, generally decreases in later iterations. So for a certain job, we normalize the change of loss values in the current iteration with respect to the largest change we have seen so far.

Figure 3.5 shows the normalized changes of loss values for common ML algorithms (summarized in Table 3.1). Because a loss function eventually converges to a certain value, the corresponding change of loss values always converges to 0. As a result, even though the set of algorithms have diverse loss ranges, we observe that they generally follow similar convergence properties, and can be normalized to decrease from 1 to 0. This helps SLAQ track the progress of different training jobs, and, for each job, correctly project the time to reach a certain loss reduction with a given resource allocation.

SLAQ supports a large class of important ML algorithms, but currently does not support some non-convex optimization algorithms due to the lack of convergence analytical models.

3.3.2 Measuring and Predicting Loss

After unifying the quality metrics for different jobs, we proceed to allocate resources for global quality improvement. When making a scheduling decision for a given job, SLAQ needs to know how much loss reduction the job would achieve by the next epoch if it was granted a certain amount of resources. We derive this information by predicting (i) how

many iterations the job will have completed by the next epoch (§3.3.2), and (ii) how much progress (i.e., loss reduction) the job could make within these iterations (§3.3.2).

Prediction for iterative ML training jobs is different from general big-data analytics jobs. Previous work [110, 19] estimates job's runtime on some given cluster resources by analyzing the job computation and communication structure, using offline analysis or code profiling. As the computation and communication pattern changes during ML model configuration tuning, the process of offline analysis needs to be performed every time, thus incurring significant overhead. ML prediction is also different from the estimations to approximate analytical SQL queries [123, 17] where the resulting accuracy can be directly inferred with the sampling rate and analytics being performed. For iterative ML training jobs, we need to make *online* predictions for the runtime and intermediate quality changes for each iteration.

Runtime Prediction

SLAQ is designed to work with distributed ML training jobs running on batch-processing computational frameworks like Spark and MapReduce. The underlying frameworks help achieve *data parallelization* for training ML models: the training dataset is large and gets partitioned on multiple worker nodes, and the size of models (i.e., set of parameters) is comparably much smaller. The model parameters are updated by the workers, aggregated in the job driver, and disseminated back to the workers in the next iteration.

SLAQ's fine-grained scheduler resizes the set of workers for ML jobs frequently, and we need to predict the iteration of each job's iteration, even while the number and set of workers available to that job is dynamically changing. Fortunately, the runtime of ML training—at least for the set of ML algorithms and model sizes on which we focus—is dominated by the computation on the partitioned datasets. SLAQ considers the total CPU time of running each iteration as $c \cdot S$, where c is a constant determined by the algorithm complexity, and S is the size of data processed in an iteration. SLAQ collects the aggregate

worker CPU time and data size information from the job driver, and it is easy to learn the constant c from a history of past iterations. SLAQ thus predicts an iteration's runtime simply by $c \cdot S/N$, where N is the number of worker CPUs allocated to the job.

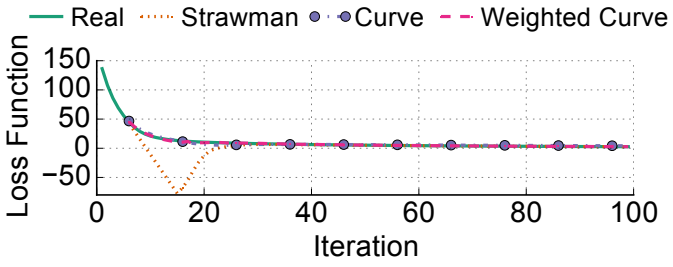
We use this heuristic for its simplicity and accuracy (validated through evaluation in §3.5.3), with the assumption that communicating updates and synchronizing models does not become a bottleneck. Even with models larger than hundreds of MBs (e.g., Deep Neural Networks), many ML frameworks could significantly reduce the network traffic with model parallelism [78] or by training with relaxed model consistency with bounded staleness [43], as discussed in §3.6. Advanced runtime prediction models [104] can also be plugged into SLAQ.

Loss Prediction

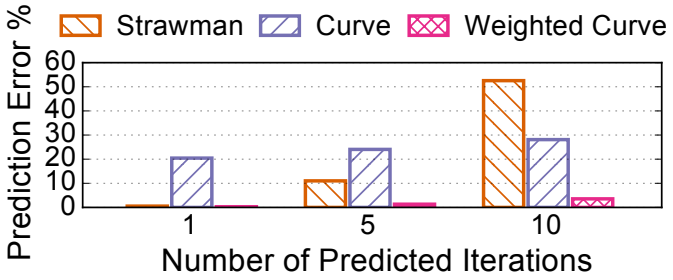
Iterations in some ML jobs may be on the order of 10s–100s of milliseconds, while SLAQ only schedules on the order of 100s of milliseconds to a few seconds. Performing scheduling on smaller intervals would be disproportionally expensive due to scheduling overhead and lack of meaningful quality changes. Further, as disparate jobs have different iteration periods, and these periods are not aligned, it does not make sense to try to schedule at "every" iteration of the jobs.

Instead, with runtime prediction, SLAQ knows how many iterations a job could complete in the given scheduling epoch. To understand how much quality improvement the job could get, we also need to predict the loss reduction in the following several iterations.

A strawman solution is to directly use the loss reduction obtained from the last iteration as the predicted loss reduction value for the following several iterations. This method actually works reasonably well if we only need to predict one or two iterations. However, this could perform poorly in practice when the number of iterations per scheduling epoch is higher. This could be the case, for example, when the training dataset is small or an abundance of resources is allocated to the job.



(a) Predicting loss values 10 iterations in advance.



(b) Average loss prediction errors when predicting 1, 5 and 10 iterations in advance.

Figure 3.6: Predicting loss values with 3 methods.

We can improve the prediction accuracy by leveraging the convergence properties of the loss functions of different algorithms. Based on the optimizers used for minimizing the loss function, we can broadly categorize the algorithms by their convergence rate.

Algorithms with sublinear convergence rate. First-order algorithms in this category[2] have a convergence rate of $O(1/k)$, where k is the number of iterations [60]. For example, gradient descent is a first-order optimization method which is well-suited for large-scale and distributed computation. It can be used for SVM, Logistic Regression, K-Means, and many other commonly used machine learning algorithms. With optimized versions of gradient descent, the convergence rate could be improved to $O(1/k^2)$.

Algorithms with linear or superlinear convergence rates. Algorithms in this category[3] have a convergence rate of $O(\mu^k), |\mu| < 1$. For example, L-BFGS, which is a widely used

[2] Assume the loss function f is convex, differentiable, and ∇f is Lipschitz continuous.

[3] Assume the loss function f is convex and twice continuously differentiable, optimization algorithms can take advantage of the second-order derivative to get faster convergence.

Quasi-Newton Method, has a superlinear convergence rate which is between linear and quadratic. It can be used for SVM, Neural Networks, and others.

Distributed optimization algorithms. Optimization algorithms like gradient descent require a full pass through the complete dataset to update the model's parameters. This can be very expensive for large jobs which have data partitions stored on multiple nodes. Distributed ML training benefits from stochastic optimization algorithms. For example, stochastic gradient descent (SGD) processes a mini-batch (samples extracted from a subset of the training data) at a time and updates the parameters in each step. The significant efficiency improvement of SGD comes at the cost of slower convergence and fluctuation in loss functions. In terms of number of iterations, however, SGD still converges at a rate of $O(1/k)$ with properly randomized mini-batches.

With the assumptions of loss convergence rate, we use curve fitting to predict future loss reduction based on the history of loss values. For the set of machine learning algorithms we consider, we use the history of loss values at a certain time to fit a curve $f(k) = \frac{1}{ak^2+bk+c} + d$ for sublinear algorithms, or $f(k) = \mu^{k-b} + c$ for linear and superlinear algorithms.

We further improve the prediction accuracy using exponentially weighted loss values. Intuitively, loss values obtained in the near past are more informative for predicting the loss values in the near future. The weights assigned to loss values decay exponentially when new iterations finish, and the parameters of the curve equations get adjusted for each prediction.

We determined the weight Λ for this process to be 0.8 experimentally. This value was arrived at with a set of ML algorithms spanning across Spark MLlib, over four datasets.

Figure 3.6 shows the loss values predicted using the different methods described above. The strawman solution works well when predicting only one iteration in advance, but degrades quickly as the number of iterations to predict increases. The latter scenario is likely because SLAQ makes a scheduling decision once every epoch, which typically spans mul-

Algorithm 4 Maximizing Total Loss Reduction

- *epoch*: scheduling time epoch
- *num_cores*: total number of cores available
- *alloc*: number of cores allocated to jobs
- *prior_q*: priority queue containing jobs and their loss reduction values if allocated with one extra core

```
1: function PREDICTLOSSREDUCTION(job)
2:     pred_loss = PREDICTLOSS(job, alloc[job], epoch)
3:     pred_loss_p1 = PREDICTLOSS(job, alloc[job] + 1, epoch)
4:     return pred_loss − pred_loss_p1
5: function ALLOCATERESOURCES(jobs)
6:     for all job in active jobs do
7:         alloc[job] = 1
8:         num_cores = num_cores − 1
9:         pred_loss_red = PREDICTLOSSREDUCTION(job)
10:        prior_q.enqueue(job, pred_loss_red)
11:    while num_cores > 0 do
12:        job = prior_q.dequeue()
13:        alloc[job] = alloc[job] + 1
14:        num_cores = num_cores − 1
15:        pred_loss_red = PREDICTLOSSREDUCTION(job)
16:        prior_q.enqueue(job, pred_loss_red)
17:    return alloc
```

tiple iterations. In contrast, as shown in Figure 3.6(b), the weighted curve fitting method achieves a low average prediction error of 3.5% even when predicting up to 10 iterations in advance.

3.3.3 Scheduling Based on Quality Improvements

With accurate runtime and loss prediction, SLAQ allocates cluster CPUs to maximize the system-wide quality. SLAQ can flexibly support different optimization metrics, including both maximizing the total (sum) quality of all jobs, as well as maximizing the minimum quality (equivalent to max-min fairness) across jobs.

Maximizing the total quality. We schedule a set of J jobs running concurrently on the shared cluster for a fixed scheduling epoch T, i.e., a new scheduling decision can only be made after time T. The optimization problem for maximizing the total normalized loss reduction over a short time horizon T is as follows. Sum of allocated resources a_j cannot exceed the cluster resource capacity C.

$$\begin{aligned} \max_{j \in J} \quad & \sum_j Loss_j(a_j, t) - Loss_j(a_j, t+T) \\ s.t. \quad & \sum_j a_j \leq C \end{aligned}$$

When including job j at allocation a_j, we are paying cost of a_j and receiving value of $\Delta l_j = Loss_j(a_j, t) - Loss_j(a_j, t+T)$. The scheduler prefers jobs with highest value of $\Delta l_j / a_j$; i.e., we want to receive the largest gain in loss reduction normalized by resource spent.

Algorithm 4 shows the resource allocation logic of SLAQ. We start with $a_j = 1$ for each job to prevent starvation. At each step we consider increasing a_i (for all queries i) by one unit (in our implementation, one CPU core) and use our runtime and loss prediction logic to get the predicted loss reduction. Among these queries, we pick the job j that gives the most loss reduction, and increase a_j by one unit. We repeat this until we run out of available resources to schedule.

Maximizing the total loss reduction targets the cost-effectiveness of cluster resources. This is desirable not only on clusters used by a single company which may have high resource contention, but potentially even on multi-tenant clusters (clouds) in which revenue could be directly associated with the total quality progress (loss reduction) of ML jobs.

Maximizing the minimum quality. Below is the optimization problem to minimize the maximum loss value (or equivalently, maximizing the minimum quality) over time horizon T. With a set of J jobs running concurrently, this scheduling policy makes sure no one is *falling behind*. We require that all loss values be no bigger than l and we minimize l.

$$\begin{aligned} \min_{j \in J} \quad & l \\ s.t. \quad & \forall j : Loss_j(a_j, t+T) \leq l \\ & \sum_j a_j \leq C \end{aligned}$$

The system quality, in this case, is represented by the loss value l of the worst job j. The only way we can improve it is to reduce the loss value of j. Our heuristic is thus as follows. We start with $a_j = 1$, and at each step we pick job $i = \arg\min_j Loss_j(a_j, t+T)$. We increase its allocation a_i by one unit, recompute $Loss_i(a_i, t+T)$, and repeat this process until we run out of resources.

Maximizing the minimum quality achieves max-min fairness in model quality. It is especially useful for ML applications that include multiple collaborative models, and the overall quality is determined by the lowest quality of all the submodels. For example, a security application for network intrusion detection should train multiple collaborative models identifying distinct attacking patterns with max-min fairness in quality.

Prioritize jobs on shared clusters. The above scheduling policies are based on the assumptions that all the concurrently running jobs have equal importance, and thus they will be treated equally when comparing their quality. This can be easily adjusted to account for jobs with different importance by adding a weight multiplier to the jobs, identically to how max-min fairness can be easily changed to weighted max-min fairness.

For example, a cluster may host experiment jobs and production jobs for ML training, and a higher weight should be assigned to jobs for production uses. With the same training progress, a job with a higher weight will get its loss reduction proportionally amplified by the scheduler compared to a normal job. Thus, high-priority jobs generally get more iterations finished with SLAQ.

Mixing ML with other types of jobs. SLAQ can also run non-ML jobs sharing the same cluster with approximate ML jobs. For non-ML jobs, the scheduler falls back to fairness or reservation based resource allocation. This effectively reduces the total capacity C available to all approximate ML jobs. SLAQ follows the same algorithms to maximize the total or minimum quality under varying resource capacity C.

3.4 Implementation

We implemented SLAQ within the popular Apache Spark framework [120], and utilize its accompaning MLlib machine learning library [86]. Spark MLlib describes ML workflow as a pipeline of *transformers*, and it provides a set of high-level APIs to help design ML algorithms on large datasets. Many commonly used ML algorithms are pre-built in MLlib, including feature extraction, classification, regression, clustering, collaborative filtering, and so on. These algorithms can easily be extended and modified for specific use cases.

The SLAQ prototype is implemented based on the Spark job scheduler. Multiple jobs place the ready tasks into task pools, which are then controlled and dispatched by SLAQ scheduler. The driver programs of ML jobs continually report their loss value information for each iteration they finish.

Token bucket. SLAQ uses a token bucket algorithm to implement the resource allocation policies described in §3.3.3. At each scheduling epoch, CPU time of all allocated cores is added to each job as *tokens*. SLAQ assigns tasks to available workers, and keeps track of how many tokens are consumed by those tasks by collecting Spark worker statistics. Tasks are throttled if the corresponding job has used up its tokens.

Running unmodified ML applications. ML *applications* written using Spark MLlib can directly run on SLAQ without any modifications. This is because SLAQ extends the underlying *optimizers* (e.g., SGD, L-BFGS, etc.) APIs to report loss values at each iteration. We cover most library algorithms provided in MLlib. Even when it is necessary to add new library algorithms, one can easily adopt SLAQ by reporting loss values using SLAQ's API. This is a one-line modification in most of the algorithms present in MLlib.

3.5 Evaluation

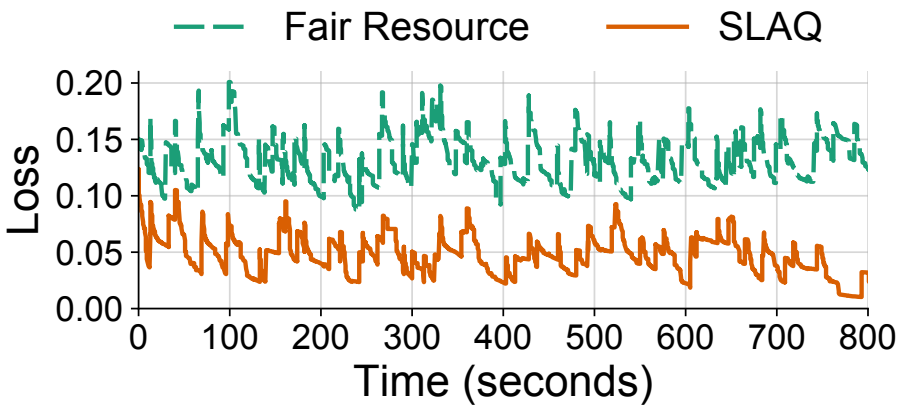
In this section, we present evaluation results on SLAQ. We demonstrate that SLAQ (i) provides significant improvement on quality and runtime for approximate ML training jobs, (ii) is broadly applicable to a wide range of ML algorithms, and (iii) scales to run a large number of ML training algorithms on clusters.

3.5.1 Methodology

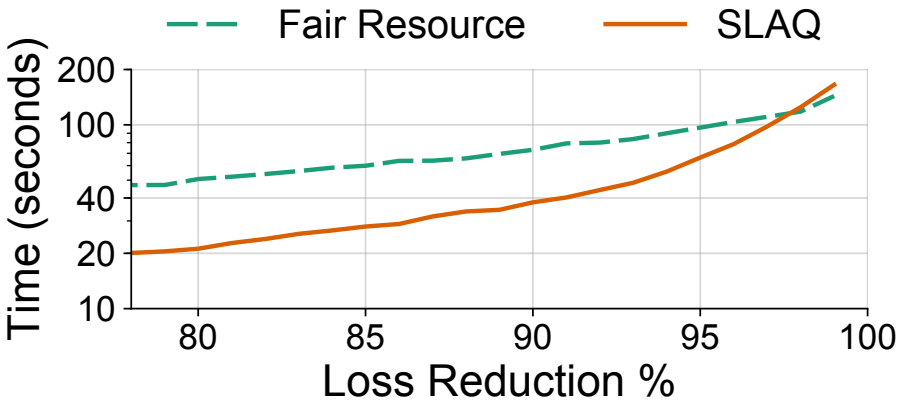
Testbed. Our testbed consists of a cluster of 20 instances of c3.8xlarge machines on Amazon EC2 Cloud. Each worker machine has 32 vCPUs (Intel Xeon E5-2680 v2 @ 2.80 GHz), 60GB RAM, and is connected with 10Gb Ethernet links.

Workload. We tested our system with the most common ML algorithms derived from MLlib with minor changes, including (i) classification algorithms: SVM, Neural Network (MLPC), Logistic Regression, GBT, and our extension to Spark MLlib with SVM polynomial kernels; (ii) regression algorithms: Linear Regression, GBT Regression; (iii) unsupervised learning algorithms: K-Means clustering, LDA. Each algorithm is further diversified to construct different models. For example, SVM with different kernels, and MLPC Neural Network with different numbers of hidden layers and perceptrons.

Datasets. With the algorithms, our models are trained on multiple datasets we collected from various online sources with modifications, as well as on our synthetic datasets. The datasets span a variety of types (plain texts [4], images [11], audio meta features [10], and so on [9]). The size of the distinct datasets we use in each run is more than 200GB. In the experiments, all the training datasets are cached as Spark Dataframes in cluster shared memory. We set the fraction of data sample processed at each iteration to be 100%, i.e., the entire training data is processed in every iteration. This choice of 100% reflects the highest level of contention we can expect to see with this many applications of these types. While



(a) Average of normalized loss values.



(b) Time to achieve loss reduction percentage.

Figure 3.7: Comparing loss improvement and runtime between SLAQ and fair scheduler.

using randomized mini-batches is common in ML applications, here they would simply introduce noise into our results.

Baseline. The baseline we compare against is a work-conserving fair scheduler. It is the widely-used scheduling policy for cluster computing frameworks [119, 63, 52, 57, 64]. The fair scheduler evenly divides available resources to all active jobs. It also dynamically adjusts resource allocations to fair share when new jobs join and old jobs leave the system.

3.5.2 System Performance

Scheduler Quality and Runtime Improvement

To evaluate job quality improvement, we first run a set of 160 ML training jobs with different algorithms, model sizes, and datasets on the shared cluster of 20 nodes. In the experiment, jobs are submitted to the cluster with their arrival time following a Poisson distribution (mean arrival time 15s). A job is considered fully converged when its normalized loss reduction is below a very small value, in this case, the loss reduction at the 100th iteration.[4] We compare the aggregate quality and runtime of these jobs between SLAQ and the fair scheduler.

Figure 3.7(a) shows the average normalized loss values across running jobs with SLAQ and the fair scheduler in an 800s time window of the experiment. When a new job arrives, its initial loss is 1.0, raising the average loss value of the active jobs; the spikes in the figure indicate new job arrivals. Yet because SLAQ allocates resources to maximize the total quality improvement (loss reduction), the average loss value of all active jobs using SLAQ is much lower than with the fair scheduler. In particular, SLAQ's average loss value is 0.49 at each scheduling epoch, which is 73% lower than that of the fair scheduler.

Figure 3.7(b) shows the average time it takes a job to achieve different loss values. As SLAQ allocates more resources to jobs that have the most potential for quality improvement, it reduces the average time to reach 90% (95%) loss reduction from 71s (98s) down to 39s (68s), 45% (30%) lower. At the very end of the job execution, further iterations take longer time as the job quality is less likely to be improved. Thus, in an environment where users submit exploratory ML training jobs, SLAQ could substantially reduce users' wait times.

Figure 3.8 explains SLAQ's benefits by plotting the allocation of CPU cores in the cluster over time. Here we group the active jobs at each scheduling epoch by their normalized loss: (i) 25% jobs with high loss values; (ii) 25% jobs with medium loss values; (iii) 50% jobs

[4]Recall that the loss reduction for each iteration is independent of the amount of resources the job is allocated; the resource allocation instead dictates the amount of *wall-clock time* each iteration takes.

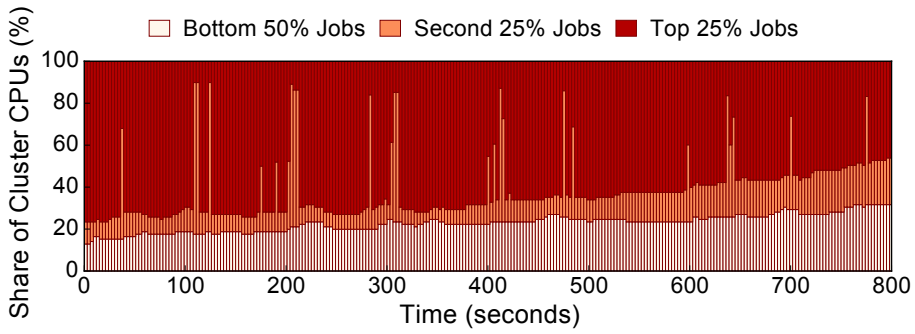


Figure 3.8: Resource allocation across jobs. At the beginning, jobs with the greatest 25% loss allocated vast majority of resources; towards the end, the difference in loss shrinks, the allocation is more spread out.

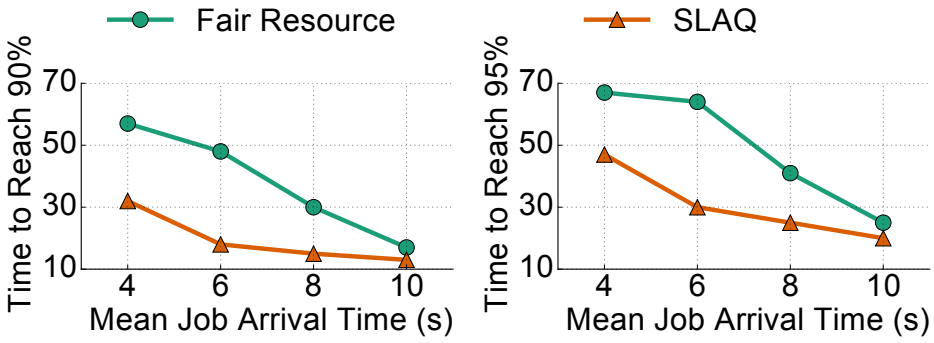


Figure 3.9: The performance difference between SLAQ and a fair resource scheduler is more significant under workloads with greater contention, e.g., jobs arriving with a mean arrival time of 4s compared to 10s.

with low loss values (almost converged). With a fair scheduler, the cluster CPUs should be allocated to the three groups proportionally to the number of jobs. In contrast, SLAQ adapts to the job quality improvement, and allocates much more computation resource to (i) and (ii). In fact, jobs in group (i) take 60% of cluster CPUs, while jobs in group (iii), despite having 50% of the population, get only 22% of cluster CPUs on average. SLAQ transfers many resources from nearly converged jobs to the jobs that have the most potential for significant quality improvement, which is the underlying reason for the improvement in Figure 3.7.

Handling Different Workloads

The achieved qualities of training jobs strongly depend on the cluster workload. As the workload increases, it becomes more important to efficiently utilize the resources. In this experiment, we vary the mean arrival time of new jobs, which in turn varies the number of concurrent jobs, and observe how SLAQ and the fair scheduler handle resource contention under different workloads.

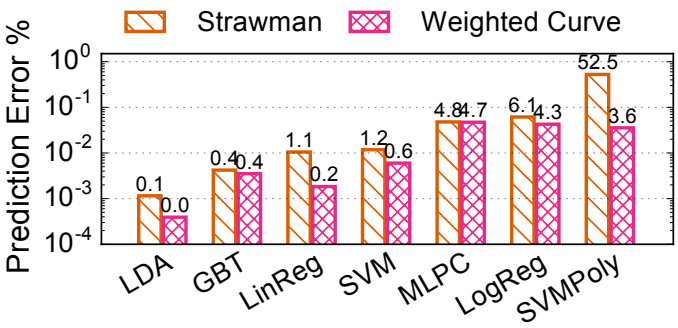
Figure 3.9 illustrates that SLAQ achieves a greater relative benefit over a fair schedule under more contentious or aggressive workloads. We start with a mean arrival time of 10s (or equivalently, 6 new jobs per minute). Under the light workload, the computation resources are relatively abundant for each job, so the time to reach 90% (95%) loss reduction is similar for both schedulers, with SLAQ performing 23% (20%) better.

As we increase the system workload with smaller mean job arrival times, cluster resource contention increases. SLAQ allocates resources to the jobs with the greatest potential. As a result, when the mean arrival time is 4s (15 new jobs per minute), SLAQ achieves an average time for jobs to reach 90% (95%) loss reduction that is 44% (30%) less than the fair scheduler.

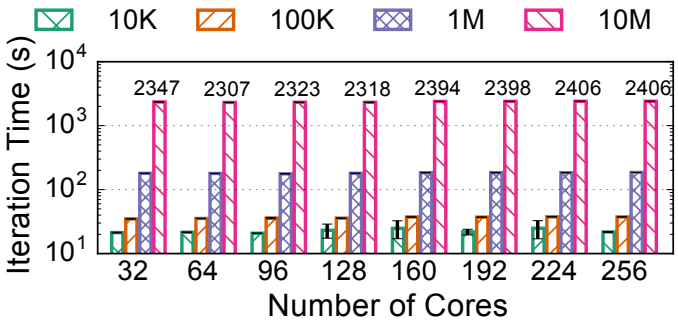
3.5.3 Robustness of Prediction

SLAQ relies on an estimate of the expected loss reduction of a job, given a certain resource alloction (see §3.3.2). To ensure stability, SLAQ makes a reallocation decision only once per scheduling epoch. Thus, the scheduler requires (i) the loss predictor to precisely estimate the loss values at least a few iterations in advance, and (ii) the runtime predictor to accurately report how long each iteration takes with a certain number of allocated cores.

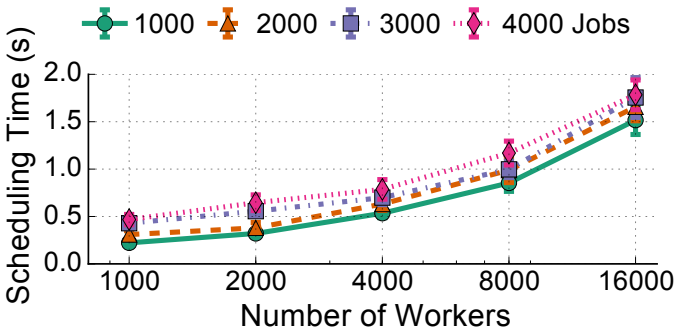
Figure 3.10(a) plots the loss prediction error for the types of ML algorithms we tested (Table 3.1). We compare the loss prediction error relative to the true values for 10 iterations, with both strawman and weighted curve fitting methods of §3.3.2. Our prediction achieves less than 5% prediction errors for all the algorithms.



(a) Predicting the next 10th iteration.



(b) Average CPU time to finish each iteration.



(c) Scheduling time.

Figure 3.10: SLAQ loss / runtime prediction and overhead.

Recall that SLAQ uses a simple heuristic to estimate the iteration runtime with N cores. To demonstrate that each iteration's CPU time is $c \cdot S$ (c as a constant), regardless of how many workers are allocated, we evaluate the total CPU time to complete an iteration with a fixed data size S. We vary the number of workers (32 cores each) between 1 and 8 and training neural network models of sizes from 10KB to 10MB. Figure 3.10(b) illustrates that, at least for ML models smaller than tens of MB, communication and model synchronization

do not affect processing time. Therefore, when dynamically changing N, an iteration's time can simply be estimated as $c \cdot S/N$. We discuss extending SLAQ to large models in §3.6.

3.5.4 Scalability and Efficiency

Figure 3.10(c) plots the time taken by SLAQ to schedule tens of thousands of concurrent jobs on large clusters (simulating both the jobs and worker nodes). SLAQ makes its scheduling decisions in between hundreds of milliseconds to a few seconds, even when scheduling 4000 jobs across 16K worker cores. These decisions are made each scheduling epoch, a timeframe of a few seconds. As shown in Figure 3.6, the more iterations in advance SLAQ predicts, the larger potential error it will incur. The agility of SLAQ enables the scheduler to predict only a few iterations in advance for each ML training job, adjusting its resource allocation decisions frequenty to meet the jobs' quality goals. SLAQ's scheduling time is comparable to the scalability of schedulers in many big data clusters today, leading us to conclude that SLAQ is sufficiently fast and scalable for (rather aggressive) real-world needs.

3.6 Discussion

Communication overhead. SLAQ is tested with ML models that have a moderate number of parameters. Recent developments in distributed frameworks for training ML models, especially deep neural networks (DNN), incur more communication and synchronization overhead between the ML job driver and worker nodes. For example, with a large number of perceptrons and multiple layers, a DNN model can grow to tens of GBs [76, 90].

Since our current implementation is based on Spark, the driver essentially becomes a single-node parameter server [3], which is responsible for gathering, aggregating, and distributing the models in every iteration. This communication overhead—due to Spark's architecture—limits our ability to train large models.

Several solutions have been proposed to mitigate the communication overhead problem. Model parallelization using architectures based on parameter servers or graph computing

proportionally scale the model serving nodes with the workers [15, 5, 78, 116]. With these optimized frameworks, SLAQ's performance improvement based on online prediction and scheduling heuristics should apply to large ML models.

Distributed ML training with relaxed consistency. Distributed ML frameworks used in practice leverage a relaxed consistency model with bounded staleness [43] to reduce the communication costs during model synchronization. The convergence progress of the underlying ML training algorithms is typically robust to a certain degree of fluctuation and slack, so the efficiency improvement obtained from the parallelism outweighs the staleness slowdown on convergence rate.

A commonly used execution model with bounded staleness is Bulk Synchronous Parallel (BSP), which allows multiple workers to individually update on partitioned training data and only synchronizes their models every several iterations [38, 116, 89]. We can extend SLAQ to support these frameworks by collecting the batch iteration time on each worker, and the model quality and communication time at each synchronization barrier to help estimate the loss reduction under the two levels of iterativeness. In fact, the convergence property of ML training is also studied in [89] with the BSP execution model under various conditions (e.g., varying communication latency and cluster sizes).

Non-convex optimization. SLAQ's loss prediction is based on the convergence property of the underlying optimizers and curve fitting with the loss history. Loss functions of non-convex optimization problems are not guaranteed to converge to global minima, nor do they necessarily decrease monotonically. The lack of an analytical model of the convergence properties interferes with our prediction mechanism, causing SLAQ to underestimate or overestimate the potential loss reduction.

One solution to this problem is to let users provide the scheduler with hint of their target loss or performance, which could be acquired from state-of-the-art results on similar problems or previous training trials. The convergence properties and optimization of non-

convex algorithms is being actively studied in the ML research community [30, 75]. We leave modeling the convergence of these algorithms to future work.

3.7 Related Work

Approximate computing systems. Many systems [25, 62, 66, 17, 21, 109] allow users to get approximate results with significantly reduced job completion time. Online aggregation databases [123, 62] generate approximate results and iteratively refine the quality. While we designed SLAQ for iterative ML training jobs, our techniques are broadly applicable to scheduling data analytics systems that iteratively refine their results.

Scheduling ML systems. Large-scale ML frameworks [86, 15, 5, 78, 37, 102, 14] optimize the computation and resource allocation for multi-dimensional matrix operators within a training job. These systems greatly accelerate the training process and reduce job's synchronization overhead. As a cluster scheduler, SLAQ could support different underlying ML frameworks (with modifications) in the future, and allocate resources at the job level to optimize across different ML training jobs.

ML model search. Several systems [103, 104] are designed to accelerate the model searching procedure. TuPAQ [104] uses a planning algorithm to discover hyperparameter settings and exclude bad trials automatically. SLAQ is designed for ML training in general exploratory settings on multi-tenant clusters. Automated model search systems could work in conjuction with SLAQ for faster decisions and better cluster utilization.

Cluster scheduling systems. Existing cluster schedulers [119, 63, 52, 27, 57, 64] primarily focus on resource fairness, job priorities, cluster utilization, or resource reservations, but do not take job quality into consideration. They mostly ignore the quality-time trade-off, and the quality trade-off between jobs. This trade-off space is crucial for ML training jobs to get approximate results with much less resource usage and lower latency.

Estimation of resource usage and runtime. Ernest [110] predicts job quality and runtime based on the internal computation and communication structures of large-scale data analytics jobs. CherryPick [19] improves cloud configuration selection process using Bayesian Optimization. Despite the generality, these systems require jobs to be analyzed offline. When users debug and adjust their models, the computation structure is likely to change very often, and thus the offline analysis will bring significant overhead. NearestFit [40] provides a progress indicator for a broad class of MapReduce applications with online prediction. SLAQ uses also online prediction to avoid offline overhead, and leverages the iterative nature of ML training jobs to improve the accuracy of prediction.

Deadline-based scheduling. Many systems [20, 111, 71, 46] utilize scheduling to meet deadlines for batch processing jobs or to reduce lag for streaming analytics jobs. Jockey [51] uses a combination of offline prediction and dynamic resource allocation to ensure batch processing queries meet their latency SLAs while minimizing their impact on other jobs sharing the cluster. Instead of hard deadlines, some real-time systems [112, 65] use soft deadlines and penalize additional delay beyond the deadlines. However, these systems mainly consider the quality-runtime trade-offs for a single job, instead of optimizing across multiple approximate jobs.

Utility scheduling. Utility functions have been widely studied in network traffic scheduling to encode the benefit of performance to users [69, 73, 80]. Recent work on live video analytics [124] leverages utility-based scheduling to provide a universal performance measurement to account for both quality and lag.

3.8 Conclusion

We present SLAQ, a quality-driven scheduling system designed for large-scale ML training jobs in shared clusters. SLAQ leverages the iterative nature of ML algorithms and obtains

application-specific information to maximize the quality of models produced by a large class of ML training jobs. Our scheduler automatically infers the models' loss reduction rate from past iterations, and predicts future resource consumption and loss improvements online for subsequent allocation decisions. As a result, SLAQ improves the overall quality of executing ML jobs faster, particularly under resource contention.

Chapter 4

ReLAQS: Reducing Latency for Approximate Queries via Scheduling

Modern web dashboards allow users to interact with data visually. Data scientists and engineers across many sectors—finance, manufacturing, communications, marketing, IoT, and DevOps—use these dashboards to quickly answer questions about their data, often either by directly writing SQL queries or using visual editors that automatically construct the appropriate SQL query. As data sizes have grown dramatically in recent years, both researchers and industry have sought ways to maintain *interactive* responsiveness, which necessitates keeping query response times sufficiently low while maintaining high quality, meaningful results.

Several approaches have been proposed to improve interactive latency when accessing very large datasets. One of the most popular has been Approximate Query Processing (AQP), which can provide quick, approximate results to users by running queries on a subset of the overall dataset.

Online aggregation In particular, online aggregation is a type of online sampling in which results are iteratively improved by progressively sampling a larger and larger percentage of the overall dataset until either the user is satisfied with the result, or the entire dataset is processed. Typically, online aggregation systems also calculate error bounds

with a given confidence, allowing the user to make a decision about whether to continue processing data.

Online aggregation has been adopted in distributed settings to further reduce latency. In these systems, each mini-batch is run sequentially, but is partitioned across many worker nodes. These multi-tenant clusters are often shared between data scientists, analysts, or applications submitting several queries simultaneously, typically on shared datasets. Though parallelizing computation this way helps reduce latency, the benefits are reduced as the clusters become bogged down with many simultaneous queries competing for the same resources. In fact, some data-science-heavy companies have reported resource requests to their query processing systems that are 5x their system capacity during peak hours [68]. When these systems become overloaded, solutions involve putting queries into long scheduling queues, causing spikes in latency. These systems become overloaded because as each new query is submitted to the cluster, all active queries' shares of resources are reduced from $\frac{1}{n}$ to $\frac{1}{n+1}$, according to the standard policy of fair-resource scheduling.

Online aggregation produces diminishing returns Each progressive sampling of the data (called a mini-batch) processes roughly the same amount of data, as they are each of approximately the same size. However, not all mini-batches within a query are equally valuable to the user. For example, upon the completion of the first mini-batch, the user goes from no knowledge of the final result to a very rough estimation. By contrast, the final mini-batch takes the user from what is already a very precise estimation to the true final result. In between, the value of each mini-batch to a user is not linear. In Figure 4.1, a query's absolute error (the difference between the estimated and true result, normalized) is shown over time as more and more mini-batches are completed. The amount of progress towards zero absolute error is not linear; due to statistical closed-form estimations of error, we can expect this curve to be sub-linear with high probability.

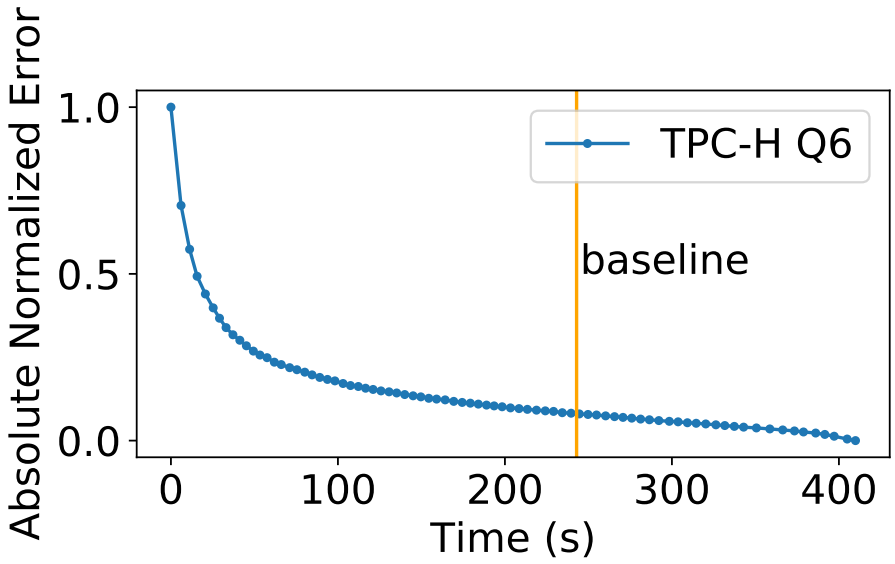


Figure 4.1: Normalized absolute error of an approximate query processing system running TPC-H query 6. The baseline here shows how long the same query takes in a traditional SQL-on-hadoop system.

This introduces a big opportunity: mini-batches that provide more valuable results to users generate more progress per unit of resource than mini-batches providing only small improvements in results. In particular, queries that have been submitted more recently will typically be improving their results more quickly than older queries. An example of this can be seen in Figure 4.2, where we show that the average progress of all queries in a cluster increases more quickly when more resources are given to queries with more potential for improvement.

Though there has been a lot of work on sampling that has focused on the trade-offs between online and offline sampling and how to best reduce error in intermediate results [17, 53, 123, 62], none of this work has addressed how to schedule approximate queries to avoid wasting resources. Due to the unique structure of these approximate queries and the multi-tenant environments in which they are being run, we argue that resources are not being apportioned correctly to help give the best results to users with minimum latency.

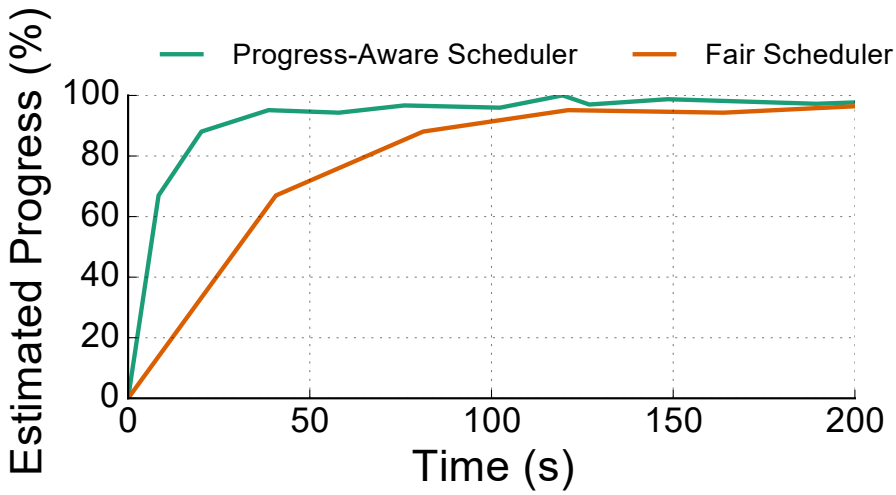


Figure 4.2: Giving more resources to queries with more error helps them get higher quality (i.e., less erroneous) results faster.

In this chapter, we take advantage of this misalignment of resources to reduce the expected latency for a query to reach a reasonably accurate result. Our scheduler uses the intermediate results provided by an online aggregation system, makes decisions about which queries have the most potential for result improvement in the near future, and reapportions resources appropriately.

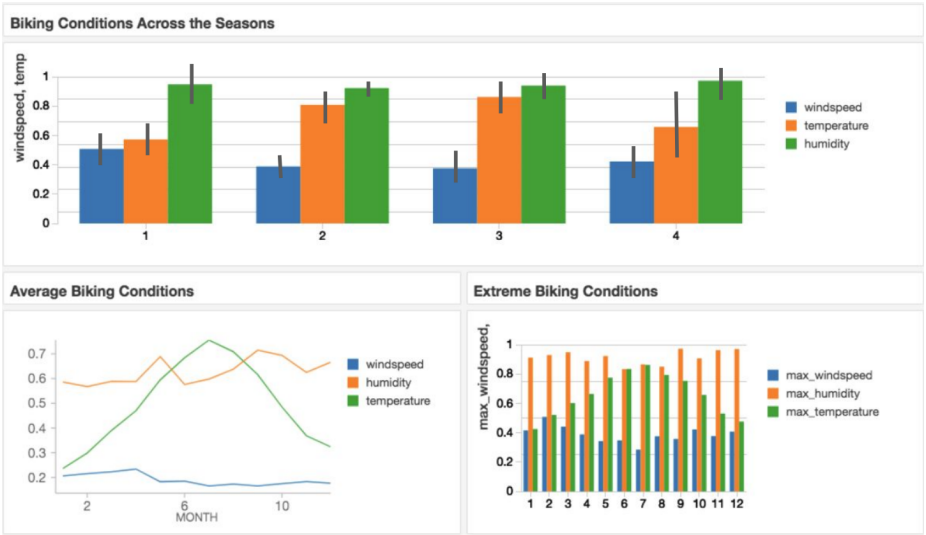


Figure 4.3: This screenshot shows an example of an interactive web dashboard that allows users to submit online SQL queries on their data.

Progressive Visualization One motivating application area for AQP systems is progressive visualization, or incremental view maintenance [18, 99]. Progressive visualization applications are typically dashboards containing one or more queries. When a page is loaded, or an action is taken by a user, those queries are submitted to an AQP system. An example of this can be seen in the sample dashboard in Figure 4.3. Because the system provides intermediate results, the progressive visualization application can begin to graph those results much earlier than dashboards backed by traditional SQL-on-Hadoop systems. Because these intermediate results are only approximate, the graphs typically display error bars to indicate how accurate the results are, and as the query continues to run on the AQP system, the graph(s) that the user sees are updated in real time.

As a motivating application for AQP systems, understanding how the latency of the underlying AQP system affects users is crucial for creating a scheduler whose goal is to reduce a user's effective latency. Prior work [99, 94] has discussed the idea that extremely small reductions in error are not useful to users who cannot see them. That is, if a query's incremental progress is not large enough to create a noticeable change to the UI provided to the user, that progress was useless, and the resources taken by that progress would have been better spent on other queries. Motivated by the same philosophy, ReLAQS gives resources to queries that are more likely to create a noticeable change to the user sooner.

Our solution This chapter introduces ReLAQS, a progress-aware scheduler. ReLAQS takes resources from older jobs whose potential for progress has slowed and gives them to newer jobs. In doing so, ReLAQS is able to improve approximate result quality and reduce latency. In designing ReLAQS, we made several important contributions.

First, we had to decide how to best quantify how much progress a particular approximate query was making compared to another. While many approximate systems provide confidence intervals to help users understand how much progress a query has made, we explore why it is impractical to use them to estimate progress in a way that allows queries

over different columns and datasets to compare to one another (§4.1.2). We propose an alternative solution using only the approximate results to compare queries' progress to one another with almost no overhead (§4.3.3).

Second, in order to predict the amount of progress an extra unit of resources provides, we must be able to predict the rate at which each query progresses with relative precision. We not only must predict the amount of error reduction each mini-batch provides, but also the amount of total compute time required for each mini-batch. This can be challenging for queries whose results are particularly noisy. We present an approach which allows us to fit a curve to the progress of a query in a responsive online manner (§4.3.4).

Third, we discovered trade-offs between traditional fair resource scheduling and ReLAQS. ReLAQS greatly reduces latency for queries in their earlier stages, and increases latency for those in very late stages. We discuss some factors affecting this trade-off, such as job arrival time and minimum resource allotment which would allow the cluster manager to favor queries at different error levels based on the needs of the users and cluster utilization (§4.5.2).

ReLAQS was implemented as a scheduler on top of Apache Spark [120] for iOLAP [123], an online aggregation system. ReLAQS supports all of the queries supported by iOLAP with no modifications. When an iOLAP query completes a mini-batch, it reports its most up-to-date estimation of the true answer to ReLAQS, which uses that information to provide resources to the various queries in the system. We evaluated ReLAQS on a subset of TPC-H benchmark queries using data sets of varying sizes provided by TPC-H. ReLAQS reduces the latency the required for the average query to reach 90% accuracy by up to 47%.

4.1 Background

In this section, we compare the various forms of AQP (§4.1.1) to illustrate why we chose to build a scheduler for incremental, sampling-based AQP systems (also known as online aggregation). We then describe the limitations of well-known error estimation methods

(§4.1.2) and how they pose a challenge for our scheduler. Finally, we have a look at the state-of-the-art cluster scheduling systems and discuss why there is a need to replace existing schedulers, which are wasteful when scheduling for AQP applications (§4.1.3).

4.1.1 Various forms of AQP

Existing AQP systems are designed along many axes, of which two are particularly important: (1) sampling vs sketches, and (2) one-time vs incremental processing. In this section, we will compare the pros and cons in both dimensions and discuss why we choose to build a scheduler for incremental, sampling-based AQP systems in our work.

Sampling and sketching are both techniques to achieve close approximation of the true answers with significant cost savings. Sampling primarily saves computation by skipping over a large fraction of the dataset, while sketching primarily saves space by maintaining lossy summaries of the data processed so far. There are two main limitations of sketching solutions: first, they are often highly-tailored to specific problems and thus not general enough to support many workloads, and second, they can never reach the true answer, since sketches are lossy by definition. For these reasons, sampling is by far the more common approach adopted in AQP systems.

Another design choice for AQP systems is whether to return a single answer at the end of computation (one-time) or to return multiple answers that are progressively refined over time (incremental). One-time processing solutions often require the user to specify a time or error bound, which may be cumbersome to provide. In contrast, incremental processing provides a smooth trade-off between computation time and query accuracy. The system can process more and more data until the user is satisfied with the answer. For example, if the user later decides they want an exact answer, then they can run the existing computation to completion (until 100% accuracy) instead of having to re-run the query from scratch.

One important advantage of online aggregation (incremental processing) is that it is more suitable in interactive settings since it gives the user quicker feedback. This is important because one can deduce the progress of a query processing job quickly based on

the error returned with the answers. In a multi-tenant environment, the cluster scheduler can then use this information to dynamically decide how to best distribute resources among jobs running on the cluster.

4.1.2 Error estimation

A crucial feature of any AQP system is the ability to return a measure of how accurate an approximate answer is. Without this knowledge, the user cannot decide whether a given answer is good enough and whether or not to attempt to reach a more accurate answer. In this section, we discuss the shortcomings of existing error estimation techniques and why our scheduler cannot simply rely on them to measure progress in query.

Three main approaches have been proposed to estimate error: closed-form estimation [70], large deviation bounds [56], and bootstrap [50]. Closed-form estimation and large deviation bounds are both analytic methods for bounding the answer. However, because both approaches rely on manual analysis of the query operation in question, they are not applicable to general workloads, which may be arbitrarily complex. For example, one cannot use these approaches for queries containing subqueries or user-defined functions.

In contrast, bootstrap is widely applicable to general computation. This method computes confidence intervals as follows. First, given a batch of data of size n, the system repeatedly resamples this batch of data with replacement to produce many *bootstrapped* samples of the same size n. Then, the system runs a trial (computation) on each of these bootstrapped samples to produce a distribution of approximate answers. Finally, from this distribution, one can expand from the median value in both directions to find the bounds that capture 95% of the data points in this distribution (for 95% confidence intervals). For instance, this is the approach used in iOLAP[123].

The bootstrap method has two weaknesses. First, its generality comes with a cost. Running more bootstrap trials increases the quality of the bounds but also increases the amount of computation needed, hence inflating response times. Second, it may produce confidence intervals that are not accurate enough for use in practice. For instance, a study of Face-

book's queries over a 7-day period reveals that close to 40% of the bounds produced with bootstrap either overestimate or underestimate the approximation error [16]. An important corollary of this is that the width of the confidence intervals returned by these AQP systems may not be an accurate representation of the progress a query is making.

As such, we find that the main approaches for error estimation proposed by previous AQP systems are not sufficient for our needs. To build a robust scheduler for AQP systems, we must find our own way to represent progress that is both general and accurate.

4.1.3 State-of-the-art Cluster Scheduling

Modern day clusters are often shared among multiple tenants. The cluster scheduler is responsible for distributing the limited set of resources to the applications run by these tenants. In this setting, an application runs one or more *jobs*. Common objectives of the cluster scheduler include maintaining high resource utilization and ensuring resource fairness among the jobs running on the cluster. For instance, dominant resource fairness [52], a generalization of max-min fairness to multiple resources (most commonly memory, CPUs and GPUs), is the most widely-adopted scheduling algorithm adopted by existing cluster schedulers [119, 63]. These scheduling systems treat jobs running on the cluster as black boxes and make decisions only based on the demands submitted by the users and the current load.

When applied to approximate applications like incremental AQP, however, this strategy forgoes opportunities to make more efficient use of the cluster resources. Because of diminishing returns, query processing jobs in their early stages benefit much more from an extra unit of resource than those in their late stages. For example, the difference between a 90%-accurate answer and a 91%-accurate answer in a late stage job may not be important or even perceivable to the user. In contrast, the difference between a wildly inaccurate answer and a 70%-accurate answer in an early stage job actually matters. In this case, taking resources away from the late stage job and giving them to the early stage job will lead to quicker insights gained by the user. This is especially true for exploratory and visualization

use cases (e.g., dashboards), where a ballpark answer is often sufficient for the user to make decisions.

In other words, we argue that fairness should be defined in terms of the *utility* gained by the user instead of resource usage. Jobs that provide to users a lot of utility should be prioritized and given more resources. This is the scheduling philosophy adopted by our system, ReLAQS. Note that utility is a concept defined by the application. Unifying all these different notions of utility across applications sharing the cluster is a key challenge we must address.

4.2 System Overview

4.2.1 Traditional Hadoop Schedulers

ReLAQS is a cluster scheduling framework for approximate queries running in a shared multi-tenant environment. We accomplish this by targeting AQP systems with long-running workers whose jobs are partitioned into small tasks. This allows quick resource allocation changes with minimal overhead. As a result, we chose to build ReLAQS for SQL-on-Hadoop AQP systems, and the scheduler was designed to replace a Hadoop scheduler.

In a traditional Hadoop cluster, one server acts as the centralized scheduler while other servers are workers that process data in an embarassingly parallel manner. The driver process serves as the gateway between the user and the cluster: when a job (in this case, a query) is submitted to the system, the driver asks the centralized scheduler for resources and partitions the job into a direct acyclic graph (DAG) of stages. The stages are further broken down into tasks, each of which runs the same computation on a different partition of the input data. This architecture is often referred to as *two levels of scheduling*, where the job-level scheduling happens on the centralized scheduler and the task-level scheduling happens on the driver.

Increasingly, users are starting to run the driver as a long-running process that also acts as the centralized scheduler [7, 12], thus blending the two levels of scheduling. This slightly different architecture has two important advantages. First, worker processes are now also long-running and shared across jobs, bypassing the overheads of launching and tearing down containers every time a job starts or finishes. Second, the scheduler now has control over which specific task each worker should run, allowing for fine-grained rebalancing of resources among jobs running on the cluster (as opposed to having to expensively preempt entire jobs in the traditional architecture). For the rest of the chapter, this is the architecture we assume, where the driver is also responsible for scheduling across jobs.

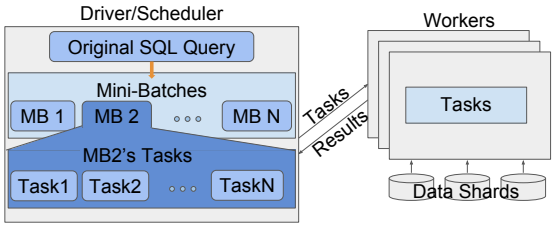
4.2.2 Online Aggregation on Hadoop

In distributed online aggregation systems, when a query is submitted to the driver, the driver randomly partitions the data into n mini-batches. These mini-batches are made up of shuffled data to ensure that each mini-batch is representative of the whole dataset. Then, as with traditional SQL-on-Hadoop systems, each mini-batch query is itself converted to a series of map and reduce stages which are further broken into tasks, each one representing the same computation over a small subset of the mini-batch. Figure 4.4(a) shows how the driver transforms the initial query to tasks for the workers to execute.

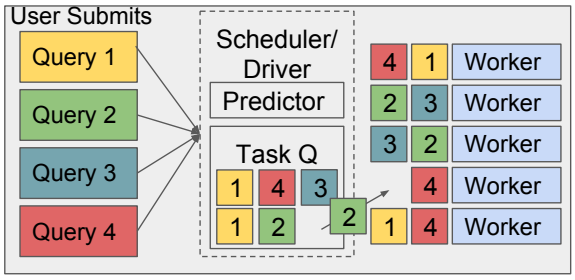
When a query's first approximate result is returned to the driver, the user is given the approximate result as well as the confidence interval surrounding that result. The user's program can then decide whether to launch the next mini-batch or accept the result and stop running the query.

4.2.3 Using AQP results to Schedule in ReLAQS

Like the fair scheduler, ReLAQS also maintains a task queue. The difference here is the task queue in ReLAQS ranks tasks based on how much potential their parent queries have on improving their results. This is shown in Figure 4.4(b).



(a) When an approximate SQL query is submitted to the driver, its data is shuffled and partitioned into mini-batches on the driver. Each mini-batch is further partitioned into tasks, which are sent to the workers.



(b) ReLAQS's task queue sorts tasks by potential progress and assigns them to the workers.

Figure 4.4: High-level ReLAQS system overview

When a mini-batch is completed, the underlying AQP system returns an approximate result to the driver. This result is forwarded to ReLAQS, which keeps a record of all approximate results of each active query. The scheduler then periodically readjusts resource allocations based on the approximate results of all of the queries in the cluster; the methods used to decide the allocations are expanded upon in section 4.3. Because each task typically is limited to between a few hundred milliseconds and a few seconds, the scheduler is able to quickly reclaim resources from one query and reapportion them to other queries in only a few seconds.

Unlike traditional Hadoop schedulers that allocate resources at the start of a job and then do not change allocations, we needed to be able to quickly reallocate as new queries enter the system or old queries become less productive. While each mini-batch (which can take a minute or two, or longer) is a separate job, adjusting resources only at the mini-batch boundary would incur a delay to resource allocation that would severely impact our ability

	Predictable	Normalizable	Low Overhead	Online	Reliably Accurate
Absolute Normalized Error	✓	✓	N/A	×	✓
Conf. Interval Width	×	✓	×	✓	×
Δ Conf. Interval Width	×	✓	×	✓	×
Absolute Result	✓	×	✓	✓	✓
Δ Absolute Result (Our Metric)	✓	✓	✓	✓	✓

Table 4.1: Defining progress is complicated; any metric must be smooth, predictable, normalizable, online, and an accurate representation of the progress an approximate query has made.

to distribute resources in a timely manner. Tasks, on the other hand, tend to have a latency on the order of $\sim 100s$ of milliseconds. This allows ReLAQS to quickly change resource allocations with extremely low overhead. Thus, ReLAQS instead reallocates resources at the task-level rather than the job-level.

4.3 Design

In this section, we discuss the key design challenges of ReLAQS and the mechanisms by which we address them. Our goal is to create a scheduler that can minimize error (alternatively, maximize progress) across all queries submitted to a multi-tenant incremental AQP system. In order to do this, we must meet a few goals. First, we must be able to accurately compare progress between queries. We propose a global metric that all queries can use to compare their progress (§4.3.1), and discuss the limitations of other approaches (§4.3.2). We must also be able to accurately predict how much a query will improve its approximate result in a given amount of time (§4.3.3). We next discuss how predicting a query's progress online is necessary to schedule resources accurately (§4.3.4) and an accurate way to do so (§4.3.5). We then discuss some queries and datasets that make prediction particularly tricky and how we addressed them (§4.3.6). Lastly, we discuss the process by which we maximize total query progress using this error prediction (§4.3.7). This algorithm is how ReLAQS decides how many resources each query will be given over the next epoch.

4.3.1 Choosing a progress metric

Measuring each approximate query's progress is a crucial part of partitioning the cluster's resource between queries. Defining a query's progress both allows ReLAQS to compare how one query's results have improved over time and how two queries' results compare to one another. In this chapter, we use the terms *progress* and *error*—the two are inversely related. A query that has made a large amount of progress has a result with a small amount of error and vice versa.

In online aggregation, after each mini-batch, an approximate result is returned to the user, along with error bounds. If ReLAQS had an oracle and knew the final answer of each query, we would know the error of each answer: simply the difference between the true answer and the estimated result so far (row 1 in Figure 4.1). However, since we do not know the final answer of a query at runtime, it is important that we estimate a query's progress online.

The metric ReLAQS uses for progress must have several important properties, enumerated in Figure 4.1. First, it must be *predictable*, i.e., it must be relatively smooth. This is necessary so that our scheduler can accurately predict how assigning resources to each query will affect their progress. Second, this metric must be *normalizable*; to compare progress between queries, we must be able to scale the progress metrics to the range [0,1]. Third, this metric must incur very *low overhead* to calculate in order to keep have a responsive scheduler with small epochs. Fourth, this metric must be known at runtime (*online*) as it will be used at runtime. Finally, it is an *accurate* predictor of the true error.

4.3.2 Limitations of Confidence Intervals

A natural way to estimate progress is to look at the width of the confidence interval(s) returned by the AQP system. After all, if a user sees a smaller confidence interval, it is natural to expect the results to be more accurate. However, using these as a metric for progress has several drawbacks.

Confidence intervals are inaccurate As discussed in §4.1, sampling-based AQP systems have taken several approaches to try to estimate error bounds. Quite a few solutions have been proposed that estimate confidence interval bounds using closed-form solutions based on the central limit theorem [17], while others have used statistical bootstrapping [123]. The former approach is only applicable to 40%-60% of queries, making it impossible to assign a progress metric to the remaining queries whose errors cannot be estimated with closed forms. Bootstrap sampling, on the other hand, only provides accurate (less than 20% error) estimations for 60%-70% of queries [16]! If we were to use the widths these confidence intervals as estimations of progress, an inaccurate estimation could lead to starvation for queries whose progress was overestimated, or wasting resources on queries whose progress was underestimated.

Confidence intervals are unpredictable In order to properly divide resources between queries, we must be able to accurately predict a query's progress. In practice, we have found through experimentation that the width of a confidence interval does not necessarily follow a predictable pattern when using statistical bootstrapping. In Figure 4.5, we see that for some queries (Queries 1,6 of the TCP-H benchmark), the normalized width of the confidence interval does not change much as more data is processed, while in others, it does (Queries 16,19). While in both cases, the width of the confidence interval is generally predictable, in the cases of Q1 and Q6, the confidence interval does not accurately estimate the absolute error of the result, nor is it comparable between queries with different data ranges.

To rectify this, we investigated using the *change* in confidence interval width, which certainly will approach 0 as more data is processed. However, we found that the change in confidence interval has the potential to be quite noisy. Moreover, online normalization doesn't work if the largest value is not in one of the first mini-batches. If a later mini-batch's confidence interval change is greater than that of the first mini-batch, our scheduler

would have overestimated progress up until that point, starving that query. Because the width of the confidence interval is so noisy for many queries, this problem happens often.

Because of the inaccuracy of predicted confidence intervals, the unpredictability of them in an online manner, and the overhead sometimes associated with them described in §4.1, it is clear that we need to use a different metric to understand how much progress a query has made.

4.3.3 Change in Estimated Result as Progress

Instead, ReLAQS uses the normalized change in the estimated result to estimate progress, as it satisfies all of these requirements. More formally, if the i^{th} mini-batch gives an approximate result X_i, the way ReLAQS quantifies progress is the the metric:

$$P_i = \frac{X_i - X_{i-1}}{max(X_i - X_{i-1})\forall i} \tag{4.1}$$

This metric is *predictable* because its curve converges at a given rate and is relatively smooth. It is *normalizable* because it takes the full range [0,1]. It requires *low computational overhead* because it requires no additional computation once given the result by the underlying AQP system. It can be calculated *online* because its only input are approximate results which are known at runtime. Lastly, it is an *accurate estimator* of error.

As the total error in the result is reduced, the rate of change of the result slows accordingly. This is shown in Figure 4.5, which compares our normalized progress metric to the normalized absolute error of an approximate result to our progress metric. For TPC-H queries, our progress metric estimates true error with only a 5.15% error on average. This figure also compares how the change in the width of the confidence interval returned by the AQP system compares to the absolute error. Though the change in confidence interval width also approximates the estimate of true error with only 7.9% error, it is noisier and thus harder to predict. In addition, for metrics in which the confidence interval changes only slightly, the CI width metric can lead to overestimation of error in later mini-batches.

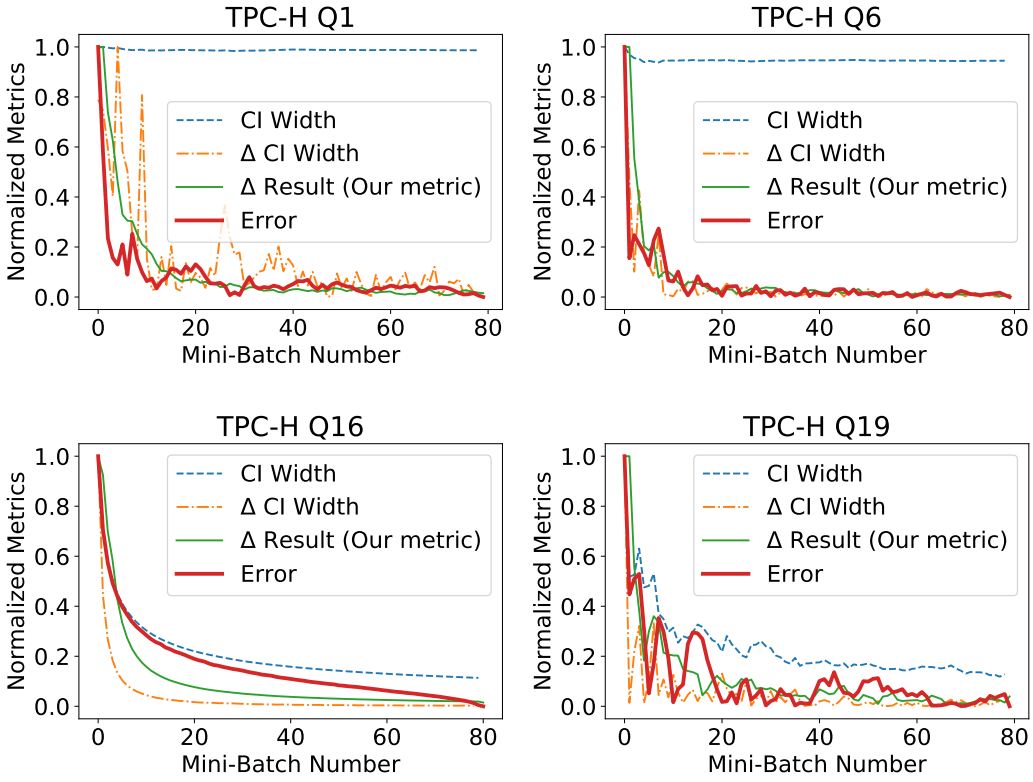


Figure 4.5: Several TPC-H queries' normalized absolute errors compared to ReLAQS's progress metric. A metric is a better estimate of progress if it more closely follows the red line labeled error.

In addition to being a good estimator for the absolute error of a query, this metric also acts as a proxy for the visual change a dashboard user would see. Because our metric measures the change in answer, we would expect the change in answer to be proportional to the change in the visual dashboard. We discuss this further in §4.6.

4.3.4 Predicting Query Progress

Now that we have an accurate and comparable progress metric P_i, we must predict how much progress each query will make over the next scheduling epoch.

At each scheduling epoch, ReLAQS predicts how much more progress each query would make if given a set of resources. This knowledge is used to help decide how many resources to apportion to each query during the next epoch. In order to make these predictions,

ReLAQS must have an analytical model to understand the rate at which the underlying AQP system reduces error in its intermediate answers.

To understand the rate at which our progress metric changes, recall that our progress metric is the normalized difference between approximate results from each mini-batch. Suppose the query is an average over a column. We call the result of the i^{th} mini-batch X_i, and our progress metric P_i (1).

Because our goal is to predict the rate of change of P_i, we can ignore our normalization term and approximate $P_i = X_i - X_{i-1}$. Our goal is to understand the curve that P_i follows as more and more tuples are processed by each subsequent mini-batch. Note that N_i, the Gaussian formed by each individual mini-batch is simply $N(\mu, \frac{\sigma}{\sqrt{n}})$.

$$
\begin{aligned}
E[P_i] &= E[X_i - X_{i-1} | X_{i-1}, ..., X_1] \\
&= E[X_i | X_{i-1}] - X_{i-1} \\
&= \frac{E[N_i]}{i} + \frac{i-1}{i} \cdot X_{i-1} - X_{i-1} \\
&= \frac{E[N_i]}{i} - \frac{1}{i} \cdot X_{i-1} \\
&= \frac{\mu}{i} - \frac{1}{i} \cdot X_{i-1} \\
&= \frac{\mu - X_{i-1}}{i}
\end{aligned} \tag{4.2}
$$

We call the absolute error $Z_i = |\mu - X_i|$ which is the difference between the true result and the estimated result after i mini-batches. The expected value of the absolute error, $E[Z_i]$, also the numerator in (1) can itself be described by the following.

$$
\begin{aligned}
E[\mu + Z_i] &= E[X_i] \\
&= \frac{E[N_i]}{i} + \frac{i-1}{i} \cdot X_{i-1} \\
&= \mu + \frac{i-1}{i} \cdot Z_{i-1} \\
\rightarrow E[Z_i] &= \frac{i-1}{i} \cdot Z_{i-1} \\
\rightarrow E[Z_i] &= \frac{Z_1}{i}
\end{aligned} \tag{4.3}
$$

Bringing (2) and (3) together, we get that $E[P_i] = \frac{Z_1}{i^2}$. Since Z_1 is a constant, and we normalize our P_i values anyways, we can predict P_i with the curve $\frac{1}{A \cdot i^2 + B}$.

We have $P_i, P_{i-1}, ..., P_1$ and want to use this information to predict future progress. We find that simply fitting the progress values we received from the active queries to this curve provided us with highly accurate predictions for the progress these queries will make during the next epoch. The accuracy of these predictions depends on how many mini-batches are expected to be completed during a single epoch.

Because ReLAQS has a very low overhead, we can keep our epochs small, as small as a few seconds. This means that our progress predictor must only predict a few seconds in the future. In our experiments, we found that even with small datasets, each mini-batch takes at minimum around one second, as the setup and synchronization overheads of each mini-batch are limited by a single driver. So even for the most extremely small datasets, it is sufficient to predict no more than 5 mini-batches in advance, and 1 mini-batch is generally sufficient.

4.3.5 Predicting Mini-batch Runtime for Nested Subqueries

While we've already decided how best to predict progress as a function of how many mini-batches have been processed, we haven't covered how we predict how much wall-time each mini-batch can be expected to use. Initially, it may seem that this is straightforward: if

each mini-batch processes the same amount of data, then we would expect each mini-batch to take the same amount of wall-time, given the same number of resources.

However, in practice, not every mini-batch will process the same amount of data. As noted in previous online aggregation work [123, 18, 122], some queries, called nested queries, are unable to process all data in a single pass. In these queries, an outer query typically depends on the result of an inner one. In traditional SQL query processors, the inner query is resolved first, but in AQP systems, an approximate result for the inner query is given, and the outer query must guess whether to process some data. If it is wrong, the AQP system must return and reprocess those tuples in later mini-batches. Therefore, later mini-batches for these queries may take longer to complete as they recompute tuples.

Given that the data is shuffled when a query is submitted, we expect the number of tuples to be recomputed to rise linearly as more and more data is computed. Other work has observed this trend [123]. This is borne out by results from previous AQP systems that show the runtime on these queries to rise linearly. If a query does not have a nested query, however, we expect each mini-batch to take the same amount of time given an equal number of resources.

ReLAQS uses past mini-batch runtimes to predict this line. By simply keeping track of how long each mini-batch takes to complete per core, we simply estimate the rate at which the mini-batches are slowing down. We use this estimated line $Ax + B$, along with the size of the mini-batch S to estimate that the $i^{\textbf{th}}$ mini-batch will take $A(\frac{i \cdot S}{N}) + B$ wall time given N workers.

4.3.6 Queries with restrictive predicates and narrow groupings

Some queries present unique challenges to understanding progress. While simpler queries may have only one approximate column, more complex queries may have multiple approximate columns, use grouping, and use filters.

Multiple Approximate Columns When a query computes multiple approximate columns, some columns may be more important than others. If a user's query has two or more approximate columns, ReLAQS must decide which column to use to compute progress. For example, consider the following query on a set of YouTube videos:

```
SELECT
    AVG(play_time), COUNT(*)
FROM
    videos
WHERE
    type='educational'
```

In this instance, it is likely that what the user is really interested in is the average play time of educational videos, and are not particularly interested in estimating the count precisely. ReLAQS provides an API allowing users to specify which approximate column(s) should be considered for the purposes of measuring progress. By default, ReLAQS assumes all approximate columns are of equal value and computes the progress as an average of each approximate column's progress. This is appropriate if a user is interested in all approximate metrics equally. Often, however, users desire a specific approximate column to converge more quickly. For example, a query submitted with two approximate columns without specifying one as being more important will result in both columns converging at a similar rate. However, if they specify that only the first approximate column is important, ReLAQS will optimize for that column. In general, this does not affect behavior much, as both columns' convergence rates are a function of what percentage of tuples have been processed.

Narrow Groupings Another potential set of problematic queries includes queries whose filters or groupings are over very small subsets. While stratified sampling [17] ensures that underrepresented groups are represented, online sampling cannot do this. While typ-

ical queries typically have enough data at the end of each mini-batch to give meaningful approximate results, these queries may complete many mini-batches without any having encountered a single tuple that either passes the filter or is a member of a narrow group. In an extreme case, consider a query like the following:

```
SELECT COUNT(*) FROM videos GROUP BY user
```

In this example, many users may have uploaded only one or two videos. The groups corresponding to those users will each have a very small amount of progress until many of the mini-batches have completed. In these cases, our progress metric would unfairly weight this particular query. It's difficult to define progress when the user has gotten no results yet, despite the fact that from a computational standpoint, many SQL rows may have been processed. One potential way to deal with these queries is to simply treat them normally. In the worst case, their progress can be difficult to predict, leading to getting a larger-than-fair share of their resources. We found that treating these queries normally in practice worked well; however, in cases with extremely long tails, it may be useful to keep track of how many rows have been computed for each group and ignore groups corresponding to too few rows.

4.3.7 Scheduling based on error reduction

Now that we have established a progress metric which can be applied to online aggregation, and can accurately predict how each query's error will be reduced in a given time, we can use this metric to maximize the progress of all of the queries in the cluster. As in resource-fair scheduling, there are several ways to optimize the utilization of the cluster depending on the optimization metric desired. While the ReLAQS system enables an operator to easily define their own custom optimization metric, we have chosen to minimize the total sum of error across all queries in the cluster as the default. Recall that our progress metric P_i allows us to compare the errors between queries due to normalization. Since our progress metric estimates the inverse of total error, this is equivalent to maximizing the

sum of the total progress (max-sum) of all queries in the cluster. This is in contrast to some traditional fair-resource schedulers [63] that focus on maximizing the *minimum* amount of progress made by any query (max-min fairness). We chose to maximize the sum of the system's progress because ReLAQS's goal is to enforce not resource-fairness, but to maximize cluster utilization.

Maximizing total progress We schedule a set of Q queries running concurrently on our multi-tenant cluster over the next scheduling epoch T. This means every T seconds, ReLAQS will recompute and redistribute resources according to new progress/error information it has received from active queries. The optimization problem for maximizing the total progress of all queries is as follows. The sum of resource allocated to query q, a_q must not exceed the total resource capacity of the cluster C.

$$\max_{q \in Q} \sum_{q} Progress(a_q, t+T) - Progress(a_q, t)$$

$$s.t. \sum_{q} a_q \leq C$$

We borrow the algorithm for this optimization from our previous SLAQ work [125], which uses the same approach to maximizing progress. This is shown in Algorithm 5.

Prioritization of performance-sensitive queries Due to the mixed settings in which queries may be submitted to the cluster, it is possible that a cluster manager may want some queries to be run with higher priority than others. For example, some queries may belong to applications providing results to users via a GUI facing customers, while some data-science-like queries may be less sensitive to high latency during periods of high cluster utilization. In these cases, ReLAQS allows queries to be submitted with a prioritization level similar to what is provided by many traditional schedulers by default. The prioritization level acts as a weight, so if a query has a prioritization level of 2, it will receive twice as

Algorithm 5 Minimizing Cluster-Wide Error

```
1: function PREDICTERRORREDUCTION(query)
2:     pred_error = PREDICTERROR(query, alloc[query], epoch)
3:     pred_error_p1 = PREDICTERROR(query, alloc[query] + 1, epoch)
4:     return pred_error − pred_error_p1
5: function ALLOCATERESOURCES(queries)
6:     for all query in active queries do
7:         alloc[query] = 1
8:         num_cores = num_cores − 1
9:         pred_error_red = PREDICTERRORREDUCTION(job)
10:        prior_q.enqueue(job, pred_error_red)
11:    while num_cores > 0 do
12:        query = prior_q.dequeue()
13:        alloc[query] = alloc[query] + 1
14:        num_cores = num_cores − 1
15:        pred_loss_red = PREDICTLOSSREDUCTION(query)
16:        prior_q.enqueue(query, pred_error_red)
17:    return alloc
```

many resources as it would have according to the optimization algorithm described above with a prioritization level of 1. This is analogous to how resource-based max-min fairness uses priorities to weight certain applications.

Mixing approximate queries with traditional queries It may be quite common that a data scientist using such a shared cluster may occasionally desire an exact solution instead of an approximate one, particularly in a case where a SQL query has no good approximable functions in it (e.g., a MEDIAN query). In this case, users are still able to submit to the same system. Due to ReLAQS's ability to quickly reallocate resources, all queries (and indeed, any jobs that may share the cluster) can be run with a fallback policy of resource-fair scheduling. After the resource-fair queries have been assigned resources, the remaining resources will be apportioned according to the progress optimization process described above.

Moreover, other applications that also can express their progress in a normalized, predictable way can share the cluster. ReLAQS will provide benefits to applications, such as machine learning tasks, whose progress also produces diminishing returns.

4.4 Implementation

ReLAQS was implemented on top of Apache Spark [120]. It is designed to be used with any iterative AQP system. In our experiments, we used iOLAP [123], an existing incremental query processing engine built on Spark SQL [24]. iOLAP automatically rewrites a subset of Spark SQL queries as delta updates, giving the user the ability to obtain partial results after only a subset of partitions have been processed. Each of these subsets is called a mini-batch, and once all mini-batches have been processed, iOLAP provides an exact result. iOLAP also provides error bounds for the approximate results via the bootstrapping method discussed in Section 4.1.

ReLAQS includes a set of modifications to the Spark job scheduler. As with traditional Spark SQL queries, when each query is submitted, a set of stages made of task pools is added to a queue, where the ReLAQS scheduler dispatches them to worker nodes. ReLAQS uses the approximate results from each query's mini-batches and uses them to make predictions and decisions about which queries should receive more resources.

ReLAQS modifies the standard resource-fair task scheduler present in Spark. When a new query is submitted to the system, ReLAQS provides it with a fair-share number of cores based on the number of queries currently in the system. Until the first mini-batch has completed, we have no information about its progress, so we simply apportion that query one $n^{\mathbf{th}}$ of the resources given n active queries in the system. It does so by assigning a priority to the tasks associated with that query. When a node in the cluster completes a task, the driver chooses a new task from its queue based on each task's priority. This priority is updated at each epoch.

Unmodified SQL Queries Because iOLAP provides approximate answers to aggregate SQL queries, ReLAQS is able to take the error bounds provided by iOLAP and schedule resources with them. In the case that the submitted query returns multiple rows (e.g., the query is an aggregate over a GROUP BY), ReLAQS simply uses the average of the error

bounds across all groups as a proxy for progress. However, if a user decides that a more specific function is a better metric for progress (e.g., the error bounds of the aggregate of one group impact progress more than another), ReLAQS provides an API by which to provide this function. iOLAP's API extends the SparkSQL API so users can submit queries as follows:

```
query = sql(YOUR QUERY HERE).online
while query.hasNext
    // get next mini-batch
    approximate_result = query.collectNext()
    display_to_user(approximate_result)
```

ReLAQS allows users to take the approximate result and notify ReLAQS of the update, as follows:

```
ReLAQS_update(approx_result, approx_columns)
```

The result here is simply the result returned by iOLAP, while *approximate_columns* is the list of columns that should be used to calculate P_i. ReLAQS uses these approximate results to modify the resource allocations.

4.5 Evaluation

In this section, we present evaluation results on ReLAQS. We demonstrate that ReLAQS (i) significantly reduces the average time for an approximate query to reach an acceptable error rate and (ii) is able to accurately predict future query error well enough to schedule queries against one another. Lastly (iii), we discuss how some tunable parameters and cluster contention affect ReLAQS's performance.

4.5.1 Methodology

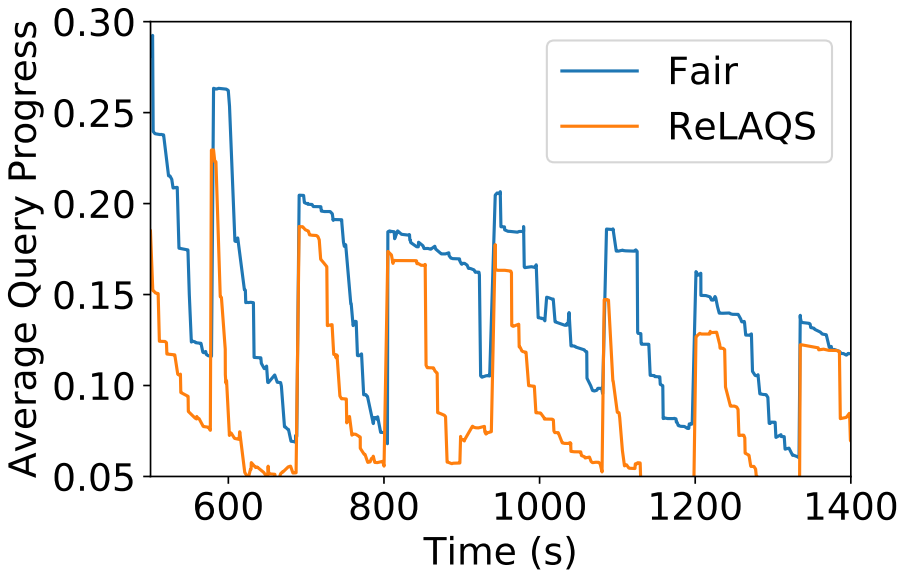
Testbed Our experimental testbed consists of a cluster of 17 servers (on which we run 1 driver and 16 workers) hosted in our university datacenter. These servers each have 16 CPU cores (Intel(R) Xeon(R) CPU E5-2670 0 @ 2.60GHz) and 60GB RAM, and they are connected with 40Gb Ethernet links.

Workload We tested our system with a subset of queries known to be supported by iOLAP[123]. This is a subset of the TPC-H benchmark (queries 1, 3, 5, 6, 11, 16, and 19). In order to diversify the set of queries, we created datasets of varying sizes up to 1 terabyte.

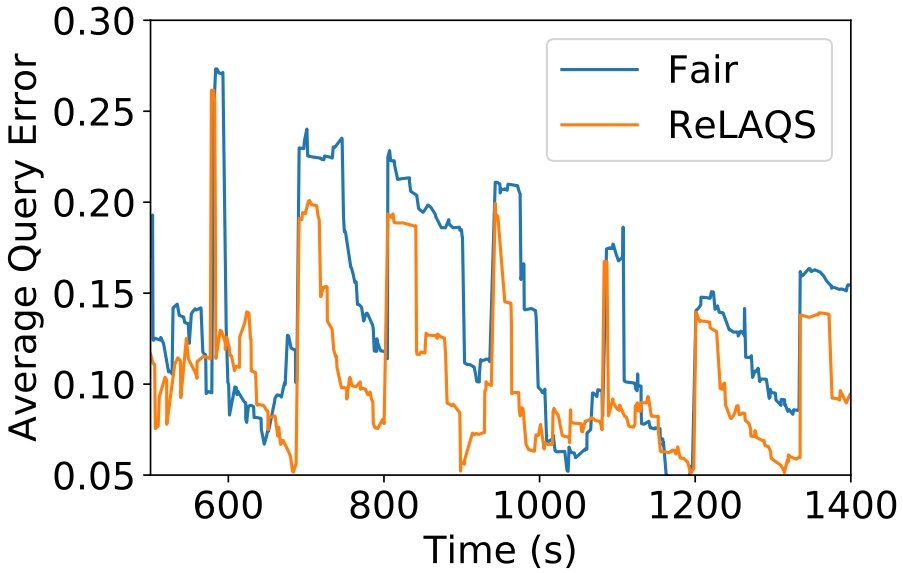
Baseline The baseline we compare against is a resource-fair scheduler. This is a popular scheduling policy cluster computing frameworks use. In fact, iOLAP was implemented on Apache Spark, which uses a resource-fair scheduler. This scheduler evenly divides resources among all active queries. When a new query joins the system or an old one leaves, it dynamically adjusts the resource allocations for all active queries.

4.5.2 Simultaneous Query Submission

To evaluate the improvement in approximate results, we first ran a set of 12 iOLAP approximate queries, representing about an hour of query execution and submission, with different TPC-H queries and varying data sizes. In this experiment, approximate queries are submitted to the cluster according to a Poisson distribution with mean arrival time of 120 seconds. This experiment simulates data scientists submitting approximate queries to a shared cluster, or alternatively, users accessing web dashboards which in turn submit approximate queries to a shared cluster. A job is considered complete when all of its mini-batches have completed, providing an exact query result to the user. We are interested in the aggregate quality of the results of these queries over time.



(a) Average Query Progress



(b) Average Query Normalized Error

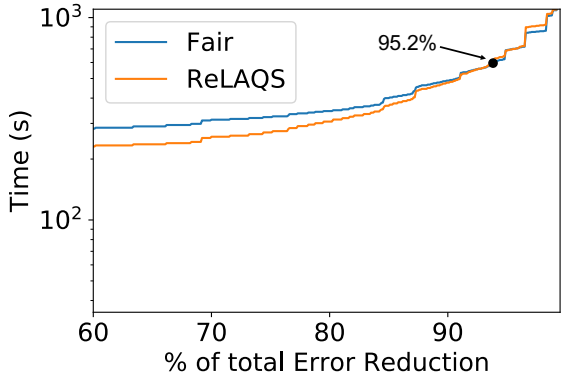
Figure 4.6: Many TPC-H queries running simultaneously. The average normalized absolute error of the estimated results of the queries currently active in the cluster is improved by ReLAQS's scheduler. For both metrics, progress and error, lower values means more accurate results. Vertical spikes indicate a new query has been submitted.

We evaluate progress via two metrics: our progress metric (the normalized change in result) and the normalized absolute error. Our progress metric tells us how quickly the estimated result is changing. As we showed in §4.3.3, the smaller the change in result, the

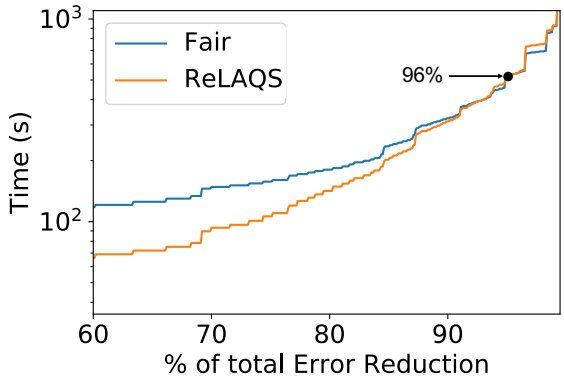
closer the query is to zero error, so smaller values mean more progress has been made. If the average change in result across all active queries is smaller, the average query in system has made more progress. As each new query is added to the cluster, the average progress metric across all queries shoots up immediately (lower indicates more progress). This is because the new query has just arrived and made no progress made so far. Then, as the cluster progresses and refines the approximate errors of all queries, the progress metric drops back down. However, for the ReLAQS scheduler, the cluster is able to focus on younger queries with the most potential for improvement. Thus, ReLAQS's average progress value drops more quickly after each query joins the cluster, as can be seen in Figure 4.6(a).

The second metric, which is similar, compares the average absolute error of the estimated results in the cluster between our scheduler and the resource-fair scheduler. In Figure 4.6(b), the same experiment has been run, but the metric being plotted on the y axis is the average normalized absolute error of each query. Because our progress metric is only an approximation for the total error of a result, it is also important to compare the absolute errors of the queries' results themselves.

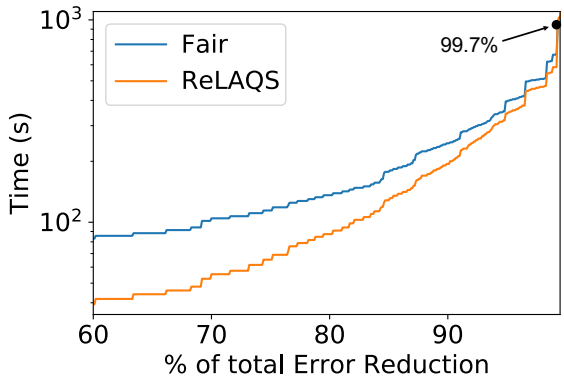
Figure 4.7, by contrast, shows the amount of time it takes for the average query in the cluster to reach a given error reduction level. Because ReLAQS allocated more resources to queries in their early stages, we see that queries scheduled with ReLAQS's scheduler reach lower absolute error more quickly. As more and more accurate results are reported, the time it takes to reach these levels approaches the amount of time it takes under the resource-fair scheduler. For example, for a query to reach 70% error reduction in Figure 4.7(c), the average query scheduled by ReLAQS takes 55 seconds compared to 104 by the resource-fair scheduler, a reduction of 47%. For a query to reduce error by 90% in Figure 4.7(c), ReLAQS reduces the average query's time from 245 down to 193, a reduction of 21%. Only if a user is waiting for an error reduction of 99.7% or more does the resource-fair scheduler outperform ReLAQS. ReLAQS can substantially reduce latency for the vast majority of approximate queries.



(a) Mean arrival interval=10 seconds



(b) Mean arrival interval=40 seconds



(c) Mean arrival interval=120 seconds

Figure 4.7: The average amount of time it takes to reach a particular error reduction with ReLAQS. Because scheduling is a zero-sum game, applications do take longer to reach error reduction above the crossing points of 95.2%, 96%, and 99.7% respectively, but significantly less time for other approximate results.

Note that this *crossing point* (i.e., the error level at which the average query's latency is equal for our scheduler and the fair-resource scheduler) varies with two variables.. The first

is a tunable parameter, the minimum amount of resources to provide to a query. Though ReLAQS provides faster results for all queries up to 99.7% of error reduction in this example, the remainder of the processing may take much longer. For example, a user who truly wants a 100% accurate result may have to wait as long as 100% longer for the final 0.3%. Essentially, ReLAQS takes some resources from these queries after they are no longer making any noticeable progress to the user. By adjusting the minimum resource allocation provided by ReLAQS, we can achieve a lower overhead for the final 0.3% while reducing the benefits provided to younger queries, lowering the crossing point.

We argue that incurring higher overheads for late stage queries for the benefit of the rest of the system is a worthy sacrifice; if the user wanted 100% accuracy, they should not be using the online aggregation system in the first place, or should submit their query as a resource-fair query. However, if the cluster manager desires, the minimum resource allocation can be increased which would lower this crossing point and reduce starvation in older queries.

In addition to minimum resource allocation, the *crossing point* is also affected by a second parameter, the utilization of the cluster. For example, if the cluster's utilization is far beyond its resources, e.g. the mean arrival interval between queries is higher than the rate at which the queries can be processed by the system, the crossing point of these two curves will be lower, as there are more queries active with high error. As the mean arrival interval approaches zero (that is, all queries are being submitted simultaneously), ReLAQS will begin to have similar latency to the fair-resource scheduler as more queries will have similar progress levels. This can be seen in Figures 4.7(a) and 4.7(b) which shows how varying the mean interval between queries affects this crossing point. With only a 10 second interval (e.g., many queries are being submitted in a quick burst), ReLAQS queries would expect to have lower latency than a fair-resource scheduler for queries up to 95.2% of total error reduction, whereas queries with an 120 second average interval would have a lower latency for queries up to 99.7%.

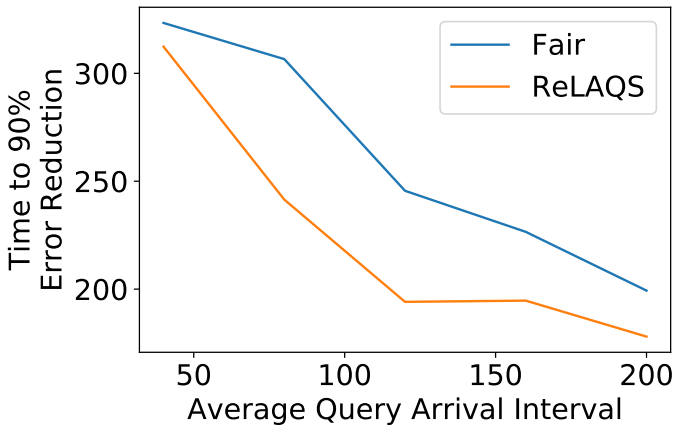


Figure 4.8: The average time a query takes to reach 90% error reduction with varying arrival frequencies.

Mean time to arrival's effect on performance Though we have chosen 120 seconds as the mean time to arrival in these experiments, it is also important to understand how the frequency with which queries are submitted affects ReLAQS. In Figure 4.8, we explore how long it takes the average query to reach 90% of error reduction with different average query submission rates.

With very high rates of submission, queries scheduled by ReLAQS are only marginally faster to reach 90% error reduction because so many queries arrive in their early stages that those around 90% receive fewer resources. As the mean time to arrival grows, ReLAQS's latency improvements over the fair resource scheduler grow, at 120 and 160 average query interval. However, as the query interval grows even further, the resource contention reduces, and ReLAQS starts to approach the same latency as the fair resource scheduler. This makes sense; a scheduler with only one active query will give all of its resources to that one query, causing equal latency between the two schedulers.

4.5.3 Prediction Accuracy

ReLAQS relies on an estimate of expected absolute error of a given query, given a certain resource allocation. To minimize total system error over the next scheduling epoch,

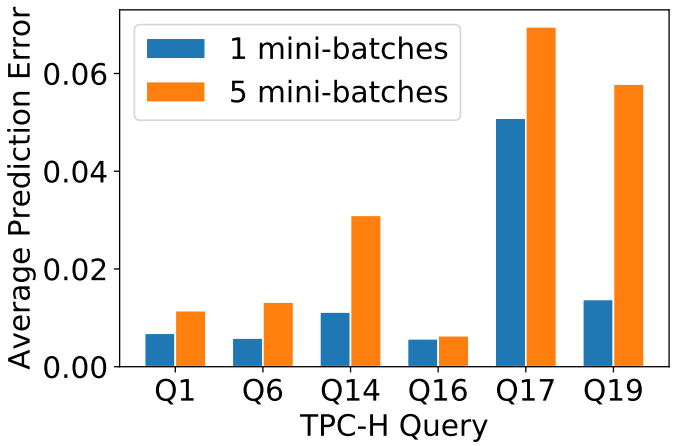


Figure 4.9: Several TPC-H queries' prediction errors. Average prediction for even noisy queries caps at 5% for 1 mini-batch in advance and 7% for 5 mini-batches in advance.

ReLAQS reallocates resources based on these predictions, so the scheduler needs the predictor to estimate the error of future mini-batches accurately.

In Figure 4.9, we take a set of TPC-H queries and use our curve-fitting techniques discussed in section 4.3. We compare the predicted values for our progress metric to the actual values tested. We were able to predict with less than 7% error for all queries tested five iterations in advance, and with less than 5% average error 1 iteration in advance. In this figure, TPC-H query 19, was a query with very narrow filters. When queries have few data points in the overall dataset that match the filters, both the results themselves and our progress metric become noisy, causing a slight rise in our predictor's error. TPC-H query 17, on the other hand, was a query with a nested query; in this example, iOLAP was forced to recompute tuples in later mini-batches because the outer query relied on the inner query, causing some tuples to have been miscomputed. In these cases, later queries sometimes have larger error in later iterations, causing noisier and harder-to-predict progress curves.

4.6 Discussion

Here we discuss some of the limitations of ReLAQS and some future work.

Non-CPU resources Due to our choice of iOLAP as our underlying AQP system, which is implemented as a set of additions to Spark SQL, we faced several implementation challenges. One of these is that Spark's scheduler (at the time of iOLAP) schedules CPU cores, while disk usage, memory, and network utilization are unaddressed. While the number of CPU cores each query gets is a good proxy for other resources, approximate query processing can be memory, disk, and network intensive.

When to stop iterative queries When queries have made significant progress, users may find it unnecessary to continue iterating as they are happy with the small error. If all users let all queries run to completion, ReLAQS can become polluted with old queries. If a user knows ahead of time the error they are willing to tolerate, they can specify that the computation should stop once the accuracy has been increased to some preset ε; this can be calculated with confidence intervals or our progress metric. While we've currently left it up to the user to stop their AQP queries, one of these approaches may be necessary to prevent the system from becoming too overloaded in a real-world scenario.

Implementation limitations Another interesting tradeoff we faced was our decision about what level we should schedule queries between each other. By deciding that we needed the ability to quickly change allocations on the order of a few seconds, Spark required that the different queries share a driver. When all queries share a driver, all active queries must share the same driver memory, a scarce resource in approximate query processing. As many queries in our benchmarks attempted to read entire tables into memory, this became a limitation. However, using multiple drivers would result in needing to kill and restart executors every few seconds, the overhead of which would outweigh the benefits provided by ReLAQS. This tradeoff is implementation specific, and could be mitigated with a different AQP system.

Progressive Visualization as a progress metric When data scientists submit queries to the system, query results are displayed to the user as progressively updating graphs. Other work has suggested that processing an additional segment of data is only useful as it affects the view displayed to the user [114]. Our progress metric, which computes the normalized difference in result, is a good approximation for this same metric. However, depending on how a user may have scaled their results, it may become necessary to use dashboard-specific information to help minimize overall cluster error. For example, in a graph where a user has plotted the approximate data on a logarithmic scale, changes in a numerically smaller range are more likely to change the output than changes in a numerically higher range. As many related works have solved the question of how to sample to best minimize visualization-based loss, applying those loss functions to ReLAQS is a promising area for future work.

Other iterative applications While this chapter focuses on iterative AQP queries, we are excited about the possibility of having ReLAQS work on a cluster sharing different types of iterative applications. Obviously, we discussed in a previous chapter ML, but we believe this can be applicable to other applications. If ReLAQS can support different applications together on the same cluster, the potential for increasing cluster utilization grows enormously.

4.7 Related Work

Approximate Query Processing systems Many systems [25, 62, 66, 17, 21, 109, 122] allow users to get approximate results with significantly reduced job completion time. Online aggregation databases [123, 62, 18] generate approximate results and iteratively refine the quality. The structure of these AQP systems shaped decisions about how ReLAQS chooses to compare and predict competing queries.

Cluster scheduling systems Existing cluster schedulers [119, 63, 52, 27, 57, 64] primarily focus on resource fairness, job priorities, cluster utilization, or resource reservations, but do not take job progress rates into consideration. They ignore the error-time trade-off, and the progress trade-off between jobs (in this case, queries). This trade-off space is crucial for AQP queries to get approximate results with lower latency.

Estimation of Accuracy Existing work to create confidence intervals around intermediary results use either bootstrap [50] or closed-form solutions [70]. Further work to estimate these methods' accuracies [16] has quantified their behavior. This work helped define ReLAQS's progress metric and thus helped predict future error for scheduling.

Approximate Query Processing Visualization As web dashboards are a major motivation for ReLAQS, exploring AQP in their context is important. Existing work has explored the utility users derive from confidence intervals of varying widths [99], as well as the comparative utilities of different visualizations [94]. They have also created loss functions meant to maximize graph accuracy with minimal sampling [95]. These loss functions can be used to map a graph's visualization to progress.

Deadline-based scheduling Many systems [20, 111, 71, 46] utilize scheduling to meet deadlines for batch processing jobs or to reduce lag for streaming analytics jobs. Jockey [51] uses a combination of offline prediction and dynamic resource allocation to ensure batch processing queries meet their latency SLAs while minimizing their impact on other competing jobs. Instead of hard deadlines, some real-time systems [112, 65] use soft deadlines and penalize additional delay beyond the deadlines. However, these systems mainly consider the quality-runtime trade-offs for a single job, instead of optimizing across multiple approximate queries.

Utility scheduling Utility functions have been widely studied in network traffic scheduling to encode the benefit of performance to users [69, 73, 80]. Recent work on live video analytics [124] leverages utility-based scheduling to provide a universal performance measurement to account for both quality and lag. In addition, our recent work on scheduling machine learning applications leverages utility-based scheduling to reduce lag [125]. We borrowed the optimization algorithm from this work, as well as some intuition about scheduling granularity.

4.8 Conclusion

We present ReLAQS, a scheduling system designed for distributed approximate query processing jobs in the environment of a shared cluster. ReLAQS leverages the diminishing returns of online aggregation to minimize the total amount of error for all active queries in a cluster. Our scheduler requires no modification to the underlying incremental AQP system. By using only the approximate results returned by the AQP system, it can conform to a wide range of online aggregation systems. ReLAQS automatically predicts future queries' progress and apportions resources accordingly. This allows it to greatly reduce latency for the vast majority of approximate queries in resource-constrained clusters.

Chapter 5

Conclusion

5.1 Summary of Contributions

This dissertation contributes techniques to achieve higher utilization of limited cluster resources for distributed clusters running approximate applications. By leveraging application-specific properties, which we identify in several applications, we demonstrated the design and implementation of cluster resource schedulers that help the cluster to produce more *useful* work for users. We evaluated these systems and showed significant latency and accuracy improvements in these applications.

More specifically, this dissertation:

- Motivates the need for utility-based scheduling for approximate applications, laying out a set of properties of certain applications that can be leveraged to achieve higher cluster-wide accuracy.
- Designs and implements SLAQ, a cluster scheduler for Machine Learning training applications that optimizes for the overall quality of models trained by a cluster simultaneously. SLAQ presents a unifying metric for utility, predicts each job's future utility, and redistributes resources using optimization to get the highest total job quality across the cluster.

- Designs and implements ReLAQS, a cluster scheduler for Online Aggregation applications that optimizes for minimizing the overall error of estimated results. ReLAQS explores potential utility metrics in context of expected use-case, uses curve fitting to predict future estimated error, and redistributes resources to minimize total error. We also discuss some design tradeoffs present in both SLAQ and ReLAQS and discuss how they affect both performance and fairness.

5.2 Open Questions and Future Work

When to stop applications All of the approximate applications we looked at in this thesis allow users to stop them early if they decide further computation is not useful to them. While sometimes users have very simple desires (e.g. stop when the result is 80% accurate), they often are more complex than that. Users may not know themselves until they have a partial result how long they want to run their job for. Furthermore, they can have complicated utility functions describing how long they're willing to wait for a given amount of additional accuracy.

However, asking users to specify their utilities per result further complicates the simple API we've provided to them, and users may not be able to easily quantify their needs. Automatically deciding when to stop in a way that keeps the API simple remains future work.

Other Approximate Applications While in this thesis we only tackle Machine Learning and Online Aggregation, we are confident there are more types of applications for which these techniques will further apply. One candidate is different types of optimization. Some of these are simple as they are very similar or indeed the same as Machine Learning algorithms we discuss in Chapter 3. Others are non-convex and require techniques we have not yet figured out. In non-convex applications, our current approach would starve applications that have lots of progress to make in the future. This could be mitigated by applying a

convex hull to their utility/loss curves, but that requires advanced knowledge of their utility curve in advance, and couldn't be done online.

As data continues to grow we are confident faster, more responsive answers will be demanded, as shown by the adoption of tqdm [47]. This will create more and more applications whose properties align with those discussed in this thesis, allowing our techniques to be applied and get faster, more accurate results.

Sharing a Cluster Between Different Types of Approximate Applications Conceptually, our system designs are designed to be job level scheduler. However, for both SLAQ and ReLAQS, we implemented these schedulers at a task level. We did this to keep our scheduler lightweight and allow it to quickly move resources from one application to another. However, a job level scheduler would allow different types of approximate applications, running on different distributed platforms, to share one big cluster, further increasing the usefulness of the cluster. We are not aware of any current job-level schedulers that can provide the fine-grained resource distribution needed, the design of which would truly allow different types of applications to share a set of machines.

5.3 Concluding Remarks

These machine learning and online aggregation applications that we've discussed throughout this dissertation are becoming extremely important to help data scientists inspect and query into the huge amounts of data companies are collecting. As this data grows, finding faster ways to gain quick, approximate insights is becoming a key challenge.

In this dissertation, we present a set of applications whose usefulness diminishes over the course of their execution. We then argue that by redistributing resources as these applications progress, we can significantly improve these applications' performance. We do this by finding universal ways to inspect their progress without significantly impacting complexity from a user perspective.

We have demonstrated systems built to schedule resources for these applications, and believe the design principles we present are broadly applicable to future distributed systems.

Bibliography

[1] Amazon Web Services Cloud. Retrieved 04/20/2017, URL: https://aws.amazon.com/.

[2] Apache Hadoop. Retrieved 02/08/2017, URL: http://hadoop.apache.org.

[3] Arimo TensorSpark. Retrieved 04/20/2017, URL: https://goo.gl/SYPMIZ.

[4] Associated Press Dataset - LDA. Retrieved 04/20/2017, URL: http://www.cs.columbia.edu/~blei/lda-c/.

[5] Caffe2. Retrieved 04/20/2017, URL: https://github.com/caffe2/caffe2.

[6] Data Sketches: Fast, Approximate Analysis of Big Data. Retrieved 10/20/2016, URL: https://yahooeng.tumblr.com/post/135390948446/data-sketches.

[7] Databricks. URL: http://databricks.com/.

[8] H2O: Open Source Platform for AI. Retrieved 04/20/2017, URL: https://docs.h2o.ai.

[9] LibSVM Data. Retrieved 04/20/2017, URL: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[10] Million Song Dataset. Retrieved 04/20/2017, URL: https://labrosa.ee.columbia.edu/millionsong/.

[11] MNIST Database. Retrieved 04/20/2017, URL: http://yann.lecun.com/exdb/mnist/.

[12] Ooyala Job Server. URL: https://github.com/ooyala/spark-jobserver.

[13] PASCAL Challenge 2008. Retrieved 04/20/2017, URL: http://largescale.ml.tu-berlin.de/instructions/.

[14] PyTorch. Retrieved 04/20/2017, URL: http://pytorch.org/.

[15] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: A System for Large-scale Machine Learning. In *USENIX OSDI*, 2016.

[16] Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 481–492, New York, NY, USA, 2014. ACM.

[17] Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*, 2013.

[18] Yanif Ahmad, Oliver Kennedy, Christoph Koch, and Milos Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.

[19] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. CherryPick: Adaptively Unearthing the Best Cloud Configurations for Big Data Analytics. In *USENIX NSDI*, 2017.

[20] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive Control of Extreme-Scale Stream Processing Systems. In *IEEE ICDCS*, July 2006.

[21] Ganesh Ananthanarayanan, Michael Chien-Chun Hung, Xiaoqi Ren, Ion Stoica, Adam Wierman, and Minlan Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, 2014.

[22] Michael Anderson, Dolan Antenucci, Victor Bittorf, Matthew Burgess, Michael Cafarella, Arun Kumar, Feng Niu, Yongjoo Park, Christopher Ré, and Ce Zhang. Brainwash: A Data System for Feature Engineering. In *CIDR*, 2013.

[23] Leonardo Aniello, Roberto Baldoni, and Leonardo Querzoni. Adaptive Online Scheduling in Storm. In *Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems*, DEBS '13, 2013.

[24] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[25] Brian Babcock, Surajit Chaudhuri, and Gautam Das. Dynamic Sample Selection for Approximate Query Processing. In *ACM SIGMOD*, 2003.

[26] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *ACM SIGCOMM*, 2011.

[27] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. In *ACM SoCC*, 2013.

[28] Léon Bottou and Yoshua Bengio. Convergence Properties of the K-Means Algorithms. In *NIPS*. MIT Press, 1995.

[29] Léon Bottou and Olivier Bousquet. The Tradeoffs of Large Scale Learning. In *NIPS*, 2008.

[30] N. Boumal, P.-A. Absil, and C. Cartis. Global Rates of Convergence for Nonconvex Optimization on Manifolds. *ArXiv e-prints*, May 2016.

[31] Craig Boutilier, Relu Patrascu, Pascal Poupart, and Dale Schuurmans. Regret-based Utility Elicitation in Constraint-based Decision Problems. In *International Joint Conference on Artificial Intelligence*, 2005.

[32] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing. In *USENIX OSDI*, 2014.

[33] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization*. Cambridge University Press, 2004.

[34] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency. In *ACM SOSP*, 2011.

[35] Urszula Chajewska, Daphne Koller, and Ronald Parr. Making Rational Decisions Using Adaptive Utility Elicitation. 2000.

[36] Badrish Chandramouli, Jonathan Goldstein, and Abdul Quamar. Scalable progressive analytics on big data in the cloud. *Proc. VLDB Endow.*, 6(14):1726–1737, Sept. 2013.

[37] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. *CoRR*, abs/1512.01274, 2015.

[38] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. In *USENIX OSDI*, Broomfield, CO, 2014.

[39] Kerry G Coffman and Andrew M Odlyzko. Internet growth: Is there a "moore's law" for data traffic? In *Handbook of massive data sets*, pages 47–93. Springer, 2002.

[40] Emilio Coppa and Irene Finocchi. On Data Skewness, Stragglers, and MapReduce Progress Indicators. In *ACM SoCC*, 2015.

[41] Graham Cormode, Minos Garofalakis, Peter J. Haas, and Chris Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Found. Trends Databases*, 4(1–3), Jan. 2012.

[42] Graham Cormode and Shan Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[43] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX ATC*, 2014.

[44] Henggang Cui, Alexey Tumanov, Jinliang Wei, Lianghong Xu, Wei Dai, Jesse Haber-Kucharsky, Qirong Ho, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. Exploiting Iterative-ness for Parallel ML Computations. In *ACM SoCC*, 2014.

[45] Yann Le Cun, John S. Denker, and Sara A. Solla. Optimal Brain Damage. In *NIPS*. 1990.

[46] Carlo Curino, Djellel E. Difallah, Chris Douglas, Subru Krishnan, Raghu Ramakrishnan, and Sriram Rao. Reservation-based Scheduling: If You're Late Don't Blame Us! In *ACM SoCC*, 2014.

[47] Casper da Costa-Luis, Stephen L., Hadrien Mary, Kyle Altendorf, noamraph, Mikhail Korobov, Ivan Ivanov, Marcel Bargull, Guangshuo CHEN, mjstevens777, Matthew D. Pagel, James, Charles Newey, Todd, Staffan Malmgren, Socialery, Max Nordlund, Martin Zugnoni, Jack McCracken, Hugo, Fabian Dill, Daniel Panteleit, Alexander, Alex Rothberg, Dyno Fu, David Bau, Arun Persaud, Andrey Portnoy, Albert Kottke, and Adnan Umer. tqdm/tqdm: tqdm v4.31.1 stable, Feb. 2019.

[48] Tathagata Das, Yuan Zhong, Ion Stoica, and Scott Shenker. Adaptive Stream Processing Using Dynamic Batch Sizing. In *ACM SoCC*, 2014.

[49] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[50] Bradley Efron and Robert J Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[51] Andrew D Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *ACM EuroSys*, 2012.

[52] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[53] Inigo Goiri, Ricardo Bianchini, Santosh Nagarakatte, and Thu D Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 383–397. ACM, 2015.

[54] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[55] Peter J Haas. *Hoeffding inequalities for join-selectivity estimation and online aggregation*. IBM, 1996.

[56] Michel Habib, Colin McDiarmid, Jorge Ramirez-Alfonsin, and Bruce Reed. *Probabilistic methods for algorithmic discrete mathematics*, volume 16. Springer Science & Business Media, 2013.

[57] Capacity Scheduler. Retrieved 04/20/2017, URL: `https://hadoop.apache.org/docs/r2.4.1/hadoop-yarn/hadoop-yarn-site/CapacityScheduler.html`.

[58] Song Han, Huizi Mao, and William J. Dally. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. *CoRR*, abs/1510.00149, 2015.

[59] F. Maxwell Harper and Joseph A. Konstan. The movielens datasets: History and context. *ACM Trans. Interact. Intell. Syst.*, 5(4):19:1–19:19, Dec. 2015.

[60] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning: Data Mining, Inference and Prediction*. Springer, 2nd edition, 2009.

[61] Joseph M. Hellerstein, Ron Avnur, and Vijayshankar Raman. Informix under control: Online query processing. *Data Min. Knowl. Discov.*, 4(4):281–314, Oct. 2000.

[62] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[63] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, and Ion Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.

[64] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[65] E. Douglas Jensen, Peng Li, and Binoy Ravindran. On Recent Advances in Time/Utility Function Real-Time Scheduling and Resource Management. *IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, 2005.

[66] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable Approximate Query Processing with the DBO Engine. *ACM Transactions on Database Systems*, 33(4):23, 2008.

[67] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable approximate query processing with the dbo engine. *ACM Trans. Database Syst.*, 33(4):23:1–23:54, Dec. 2008.

[68] Jin, Li. Preemptive scheduling in mesos framework.

[69] Ramesh Johari and John N. Tsitsiklis. Efficiency Loss in a Network Resource Allocation Game. *Math. Oper. Res.*, 29:407–435, 2004.

[70] A John. *Mathematical statistics and data analysis*. Wadsworth & Brooks/Cole, 1988.

[71] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Íñigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: towards automated SLOs for enterprise clusters. In *USENIX OSDI*, 2016.

[72] Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 631–646, New York, NY, USA, 2016. ACM.

[73] F. P. Kelly, A. K. Maulloo, and D. K. H. Tan. Rate Control for Communication Networks: Shadow Prices, Proportional Fairness and Stability. *The Journal of the Operational Research Society*, 49:237–252, 1998.

[74] Jeffrey O. Kephart. Research Challenges of Autonomic Computing. In *ACM ICSE*, 2005.

[75] S. Lacoste-Julien. Convergence Rate of Frank-Wolfe for Non-Convex Objectives. *ArXiv e-prints*, July 2016.

[76] Quoc V. Le, Rajat Monga, Matthieu Devin, Greg Corrado, Kai Chen, Marc'Aurelio Ranzato, Jeffrey Dean, and Andrew Y. Ng. Building High-Level Features Using Large Scale Unsupervised Learning. *CoRR*, abs/1112.6209, 2011.

[77] Ron Levy, Jay Nagarajarao, Giovanni Pacifici, Mike Spreitzer, Asser Tantawi, and Alaa Youssef. Performance Management for Cluster Based Web Services. In *Integrated Network Management VIII*, pages 247–261. Springer, 2003.

[78] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling Distributed Machine Learning with the Parameter Server. In *USENIX OSDI*, 2014.

[79] B. Lohrmann, P. Janacik, and O. Kao. Elastic Stream Processing with Latency Guarantees. In *IEEE ICDCS*, June 2015.

[80] Steven H. Low and David E. Lapsley. Optimization Flow Control—I: Basic Algorithm and Convergence. *IEEE/ACM Transactions on Networking*, 7(6):861–874, 1999.

[81] Steven H Low and David E Lapsley. Optimization Flow Control-I: Basic Algorithm and Convergence. *IEEE/ACM Transactions on Networking*, 7(6):861–874, 1999.

[82] Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based hyperparameter optimization through reversible learning. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 2113–2122. JMLR.org, 2015.

[83] Dougal Maclaurin, David Duvenaud, and Ryan P. Adams. Gradient-based Hyperparameter Optimization through Reversible Learning. 2015.

[84] Kshiteej Mahajan, Mosharaf Chowdhury, Aditya Akella, and Shuchi Chawla. Dynamic query re-planning using QOOP. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 253–267, Carlsbad, CA, 2018. USENIX Association.

[85] Peter Marbach. Priority Service and Max-Min Fairness. In *IEEE INFOCOM*, 2002.

[86] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. MLlib: Machine Learning in Apache Spark. *CoRR*, abs/1505.06807, 2015.

[87] Merton, R.C. *Continuous-Time Finance*. Macroeconomics and Finance Series. Wiley, 1992.

[88] Manfred Morari and Jay H Lee. Model Predictive Control: Past, Present and Future. *Computers & Chemical Engineering*, 23(4):667–682, 1999.

[89] Philipp Moritz, Robert Nishihara, Ion Stoica, and Michael I. Jordan. SparkNet: Training Deep Networks in Spark. *CoRR*, abs/1511.06051, 2015.

[90] Karl Ni, Roger A. Pearce, Kofi Boakye, Brian Van Essen, Damian Borth, Barry Chen, and Eric X. Wang. Large-Scale Deep Learning on the YFCC100M Dataset. *CoRR*, abs/1502.03409, 2015.

[91] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 293–307, 2015.

[92] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 69–84, New York, NY, USA, 2013. ACM.

[93] Niketan Pansare, Vinayak R. Borkar, Chris Jermaine, and Tyson Condie. Online Aggregation for Large MapReduce Jobs. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[94] Aditya Parameswaran, Neoklis Polyzotis, and Hector Garcia-Molina. Seedb: Visualizing database queries efficiently. *Proc. VLDB Endow.*, 7(4):325–328, Dec. 2013.

[95] Yongjoo Park, Michael Cafarella, and Barzan Mozafari. Visualization-aware sampling for very large databases. In *Data Engineering (ICDE), 2016 IEEE 32nd International Conference on*, pages 755–766. IEEE, 2016.

[96] Daniel Peng and Frank Dabek. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *USENIX OSDI*, 2010.

[97] D.M.W. Powers. Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation. *Journal of Machine Learning Technologies*, 2(1):37–63, 2011.

[98] Ariel Rabkin, Matvey Arye, Siddhartha Sen, Vivek S. Pai, and Michael J. Freedman. Aggregation and Degradation in JetStream: Streaming Analytics in the Wide Area. In *USENIX NSDI*, 2014.

[99] Sajjadur Rahman, Maryam Aliakbarpour, Ha Kyung Kong, Eric Blais, Karrie Karahalios, Aditya Parameswaran, and Ronitt Rubinfield. I've seen "enough": Incrementally improving visualizations to support rapid decision making. *Proc. VLDB Endow.*, 10(11):1262–1273, Aug. 2017.

[100] Jags Ramnarayan, Barzan Mozafari, Sumedh Wale, Sudhir Menon, Neeraj Kumar, Hemant Bhanawat, Soubhik Chakraborty, Yogesh Mahajan, Rishitesh Mishra, and Kishor Bachhav. Snappydata: A hybrid transactional analytical store built on spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 2153–2156. ACM, 2016.

[101] Brian T. Ratchford. Cost-Benefit Models for Explaining Consumer Choice and Information Seeking Behavior. *Manage. Sci.*, 28(2):197–212, Feb. 1982.

[102] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*, 2016.

[103] Jasper Snoek, Hugo Larochelle, and Ryan P. Adams. Practical Bayesian Optimization of Machine Learning Algorithms. In *NIPS*, 2012.

[104] Evan R. Sparks, Ameet Talwalkar, Daniel Haas, Michael J. Franklin, Michael I. Jordan, and Tim Kraska. Automating Model Search for Large Scale Machine Learning. In *ACM SoCC*, 2015.

[105] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, 2003.

[106] Gerald Tesauro, William E. Walsh, and Jeffrey O. Kephart. Utility-Function-Driven Resource Allocation in Autonomic Systems. In *Proceedings of the Second International Conference on Automatic Computing*, 2005.

[107] T. N. Theis and H. . P. Wong. The end of moore's law: A new beginning for information technology. *Computing in Science Engineering*, 19(2):41–50, Mar 2017.

[108] Yoh'ichi Tohkura. A Weighted Cepstral Distance Measure for Speech Recognition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 35:1414–1422, 1987.

[109] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. The Power of Choice in Data-aware Cluster Scheduling. In *USENIX OSDI*, 2014.

[110] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient Performance Prediction for Large-Scale Advanced Analytics. In *USENIX NSDI*, Santa Clara, CA, 2016.

[111] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *ICAC*, 2011.

[112] Ernesto Wandeler and Lothar Thiele. Real-time Interfaces for Interface-based Design of Real-time Systems with Fixed Priority Scheduling. In *5th ACM International Conference on Embedded Software*, 2005.

[113] Kevin C. Webb, Arjun Roy, Kenneth Yocum, and Alex C. Snoeren. Blender: Upgrading Tenant-based Data Center Networking. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2014.

[114] Eugene Wu, Lilong Jiang, Larry Xu, and Arnab Nandi. Graphical perception in animated bar charts. *arXiv preprint arXiv:1604.00080*, 2016.

[115] Sai Wu, Beng Chin Ooi, and Kian-Lee Tan. Continuous sampling for online aggregation over multiple queries. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 651–662, New York, NY, USA, 2010. ACM.

[116] Wencong Xiao, Jilong Xue, Youshan Miao, Zhen Li, Cheng Chen, Ming Wu, Wei Li, and Lidong Zhou. Tux: Distributed Graph Computation for Machine Learning. In *USENIX NSDI*, 2017.

[117] Y. Xing, S. Zdonik, and J. H. Hwang. Dynamic load distribution in the Borealis stream processor. In *IEEE ICDE*, April 2005.

[118] Le Xu, Boyang Peng, and Indranil Gupta. Stela: Enabling Stream Processing Systems to Scale-in and Scale-out On-demand. In *IEEE International Conference on Cloud Engineering (IC2E)*, IC2E '16, pages 22–31, April 2016.

[119] Apache Hadoop YARN. Retrieved 02/08/2017, URL: `http://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-site/YARN.html`.

[120] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX NSDI*, 2012.

[121] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: Fault-tolerant streaming computation at scale. In *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, pages 423–438. ACM, 2013.

[122] Kai Zeng, Sameer Agarwal, Ankur Dave, Michael Armbrust, and Ion Stoica. G-ola: Generalized on-line aggregation for interactive analysis on big data. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 913–918. ACM, 2015.

[123] Kai Zeng, Sameer Agarwal, and Ion Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *ACM SIGMOD*, 2016.

[124] Haoyu Zhang, Ganesh Ananthanarayanan, Peter Bodik, Matthai Philipose, Paramvir Bahl, and Michael J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *USENIX NSDI*, 2017.

[125] Haoyu Zhang*, Logan Stafman*, Andrew Or, and Michael J. Freedman. Slaq: Quality-driven scheduling for distributed machine learning. In *Proceedings of the 2017 Symposium on Cloud Computing*, SoCC '17, pages 390–404, New York, NY, USA, 2017. ACM.

[126] Haoyu Zhang, Logan Stafman, Andrew Or, and Michael J. Freedman. SLAQ: quality-driven scheduling for distributed machine learning. *CoRR*, abs/1802.04819, 2018.

[127] Shlomo Zilberstein. Using anytime algorithms in intelligent systems. *AI magazine*, 17(3):73, 1996.