ReLAQS: Reducing Latency for Multi-Tenant Approximate Queries via Scheduling

Logan Stafman
Princeton University
loganstafman@gmail.com

Andrew Or
Princeton University
andrewor@cs.princeton.edu

Michael J. Freedman
Princeton University
mfreed@cs.princeton.edu

Abstract

Approximate Query Processing has become increasingly popular as larger data sizes have increased query latency in distributed query processing systems. To provide such approximate results, systems return intermediate results and iteratively update these approximations as they process more data. In shared clusters, however, these systems waste resources by directing resources to queries that are no longer improving the results given to users.

We describe ReLAQS, a cluster scheduling system for online aggregation queries that aims to reduce latency by assigning resources to queries with the most potential for improvement. ReLAQS utilizes the approximate results each query returns to periodically estimate how much progress each concurrent query is currently making. It then uses this information to predict how much progress each query is expected to make in the near future and redistributes resources in real-time to maximize the overall quality of the answers returned across the cluster. Experiments show that ReLAQS achieves a reduction in latency of up to 47% compared to traditional fair schedulers.

CCS Concepts • **Computer systems organization** → **Distributed architectures**; **Cloud computing**; • **Information systems** → *Database query processing*; • **Theory of computation** → *Approximation algorithms analysis*

Keywords scheduling, approximate computing, utility-aware scheduling

1. Introduction

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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

Middleware'19 December 8–13, 2019, Davis, CA, USA

© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ISBN 978-1-4503-7009-7/19/12…$15.00

DOI: http://dx.doi.org/10.1145/3361525.3361553

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 [19]. 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 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.

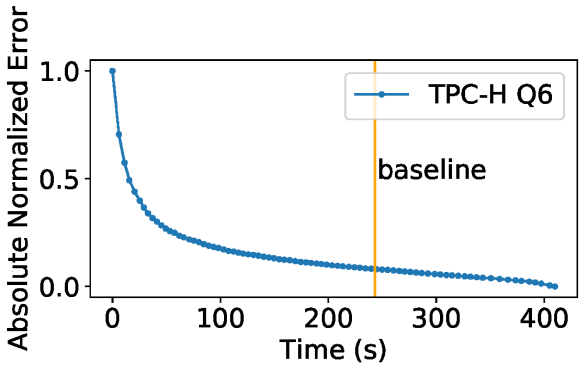


Figure 1: Normalized absolute error of an approximate query processing system running TPC-H query 6. The baseline here shows how long the same query takes in a traditional SQL-on-hadoop system.

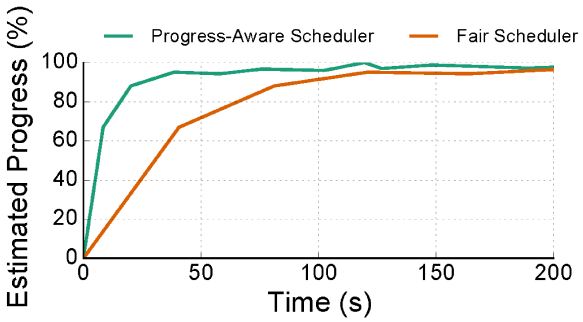


Figure 2: Giving more resources to queries with more error helps them get higher quality (i.e., less erroneous) results faster.

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 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. In fact, this approach can be taken with any applications that produce diminishing returns, such as machine learning applications [36].

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 [4, 12, 15, 34], none of this work has addressed how to schedule approximate queries to avoid wasting resources. More work still has focused on approximate applications in multi-tenant environments, but this work does so with application-specific knowledge [24, 30]. 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.

In this paper, 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.

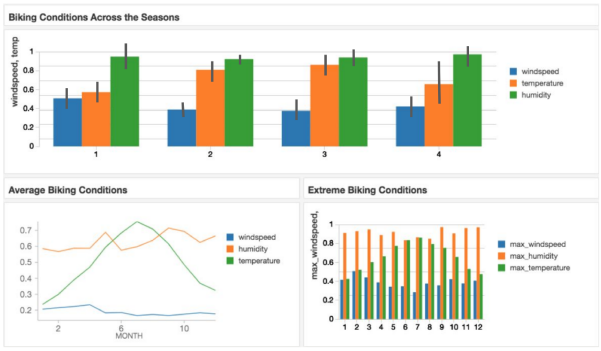


Figure 3: This screenshot shows an example of an interactive web dashboard that allows users to submit online SQL queries on their data.

Progressive Visualization One motivating application area for AQP systems is progressive visualization, or incremental view maintenance [5, 27]. 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 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 [25, 27] 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 paper 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 (§2.2). We propose an alternative solution

using only the approximate results to compare queries' progress to one another with almost no overhead (§4.3).

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 (§4.4).

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 (§6.2).

ReLAQS was implemented as a scheduler on top of Apache Spark [32] for iOLAP [34], 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%.

2. Background

In this section, we compare the various forms of AQP (§2.1) 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 (§2.2) 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 (§2.3).

2.1 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. Futhermore, incorrect time and error bounds waste computation. 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 (one form of 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. ReLAQS's techniques apply to all incremental processing systems, but we chose to focus on online aggregation in this paper.

2.2 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 [21], large deviation bounds [13], and bootstrap [10]. 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[34].

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 Facebook'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 [3]. 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.

2.3 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 [11], 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 [16, 31]. 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.

3. System Overview

3.1 Traditional Hadoop Schedulers

ReLAQS is a cluster scheduling framework for approximate queries running in a shared multi-tenant environment. As ReLAQS was built for SQL-on-Hadoop AQP systems, 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 [1, 2], 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 paper, this is the architecture we assume, where the driver is also responsible for scheduling across jobs.

3.2 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 4a shows how the driver transforms the initial query to tasks for the workers to execute.

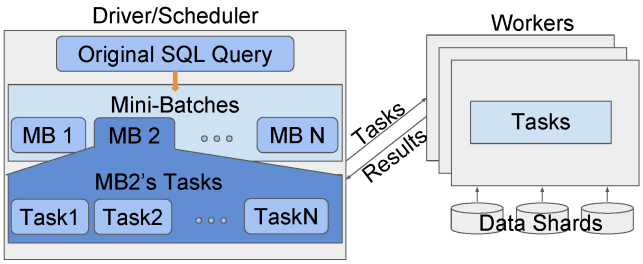
When a query's first approximate result is returned to the driver, the user is given the approximate result as well as the confidence interval surrounding that result. The user's program can then decide whether to launch the next mini-batch or accept the result and stop running the query.

3.3 Using AQP results to Schedule in ReLAQS

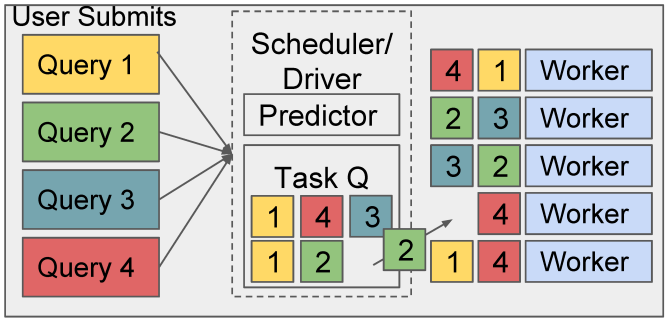
Like the fair scheduler, ReLAQS also maintains a task queue. The difference here is the task queue in ReLAQS ranks tasks based on how much potential their parent queries have on improving their results. This is shown in Figure 4b.

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. 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 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.



(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: High-level ReLAQS system overview

4. 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 (§4.1), and discuss the limitations of other approaches we explored (§4.2). 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). We next discuss how predicting a query's progress online is necessary to schedule resources accurately (§4.4) and an accurate way to do so (§4.5). We then discuss some queries and datasets that make prediction particularly tricky and how they could be addressed (§4.6). Lastly, we discuss the process by which we maximize total query progress using this error prediction (§4.7). This algorithm is how ReLAQS decides how many resources each query will be given over the next epoch.

4.1 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 paper, 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.

A strawman approach may be to simply take a query's age as a measure of progress; the longer an approximate query has been running, the more accurate the result is. While this would work if all queries converged at the same rate and process the same amount of data, the age of a query is not normalizable across different queries. For example, one query that processes 1 MB of data may have reached a very accurate result after only one second, while another that processes 1 TB will have made very little progress in that same amount of time.

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 5). 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 5. 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 must be an *accurate* predictor of the true error.

4.2 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 §2, 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 [4], while others have used statistical bootstrapping [34]. 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 [3]! 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 6, we see that for some queries (Queries 1,6 of the TPC-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 abso-

	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)	✓	✓	✓	✓	✓

Figure 5: Defining progress is complicated; any metric must be smooth, predictable, normalizable, online, and an accurate representation of the progress an approximate query has made.

lute 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.

Due of the inaccuracy of predicted confidence intervals, the unpredictability of them in an online manner, and the overhead sometimes associated with them described in §2, it is clear that we need to use a different metric to understand how much progress a query has made.

4.3 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:

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

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 6, which compares our normalized progress metric to the normalized absolute error of a 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 confidence interval width metric can lead to overestimation of error in later mini-batches.

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 §7.

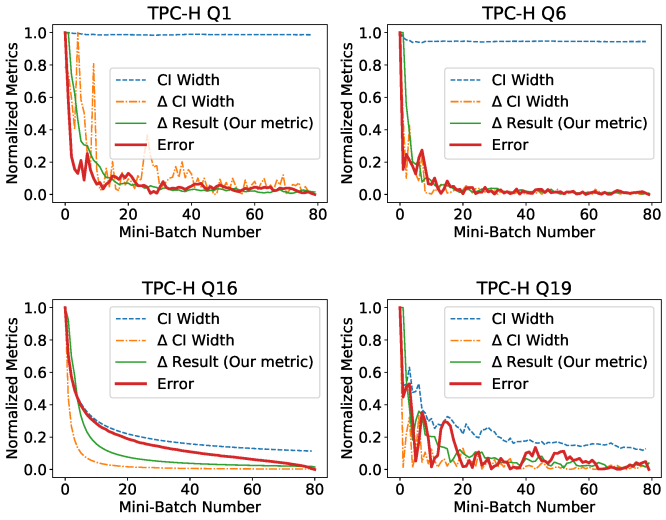


Figure 6: Several TPC-H queries' normalized absolute errors compared to ReLAQS's progress metric. A metric is a better estimate of progress if it more closely follows the red line labeled error.

4.4 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 $N(\mu, \frac{\sigma}{\sqrt{n}})$.

$$
\begin{aligned}
E[P_i] &= E[X_i - X_{i-1} | X_{i-1}, ..., X_1] \\
&= E[X_i | X_{i-1}] - X_{i-1} \\
&= \frac{E[N_i]}{i} + \frac{i-1}{i} \cdot X_{i-1} - X_{i-1} \\
&= \frac{E[N_i]}{i} - \frac{1}{i} \cdot X_{i-1} \\
&= \frac{\mu}{i} - \frac{1}{i} \cdot X_{i-1} \\
&= \frac{\mu - X_{i-1}}{i}
\end{aligned} \tag{2}
$$

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.

$$
\begin{aligned}
E[\mu + Z_i] &= E[X_i] \\
&= \frac{E[N_i]}{i} + \frac{i-1}{i} \cdot X_{i-1} \\
&= \mu + \frac{i-1}{i} \cdot Z_{i-1} \\
\rightarrow E[Z_i] &= \frac{i-1}{i} \cdot Z_{i-1} \\
\rightarrow E[Z_i] &= \frac{Z_1}{i}
\end{aligned} \tag{3}
$$

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.

4.5 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 [5, 33, 34], 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 [34]. 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^{th} mini-batch will take $A(\frac{i \cdot S}{N}) + B$ wall time given N workers.

4.6 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. In our experiments, we used this default option in order to fairly compare queries.

Narrow Groupings Another potential set of problematic queries includes queries whose filters or groupings are over very small subsets. While stratified sampling [4] ensures that underrepresented groups are represented, online sampling cannot do this. While typical 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.

4.7 Scheduling based on error reduction

Now that we have established a progress metric that 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 [16] 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 previous work [36], which uses the same approach to maximizing progress. This is shown in Algorithm 1.

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 UI facing customers, while some data-science-like queries may be less sensitive to high latency during periods of high cluster utilization. In these cases,

Algorithm 1 Minimizing Cluster-Wide Error

– $epoch$: scheduling time epoch
– num_cores: total number of cores available
– $alloc$: number of cores allocated to queries
– $prior_q$: priority queue containing queries and their error values if allocated with one extra core

function PREDICTERRORREDUCTION($query$)
  $pred_error$ = PREDICTERROR($query, alloc[query], epoch$)
  $pred_error_p1$ = PREDICTERROR($query, alloc[query] + 1, epoch$)
  return $pred_error - pred_error_p1$
function ALLOCATERESOURCES($queries$)
  for all $query$ in active $queries$ **do**
    $alloc[query] = 1$
    $num_cores = num_cores - 1$
    $pred_error_red$ =PREDICTERRORREDUCTION(job)
    $prior_q.enqueue(job, pred_error_red)$
  while $num_cores > 0$ **do**
    $query = prior_q.dequeue()$
    $alloc[query] = alloc[query] + 1$
    $num_cores = num_cores - 1$
    $pred_loss_red$ =PREDICTLOSSREDUCTION($query$)
    $prior_q.enqueue(query, pred_error_red)$
  return $alloc$

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 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.

5. Implementation

ReLAQS was implemented on top of Apache Spark [32]. It is designed to be used with any iterative AQP system. In our experiments, we used iOLAP [34], an existing incremental query processing engine built on Spark SQL [7]. 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 2.

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^{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.

6. 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.

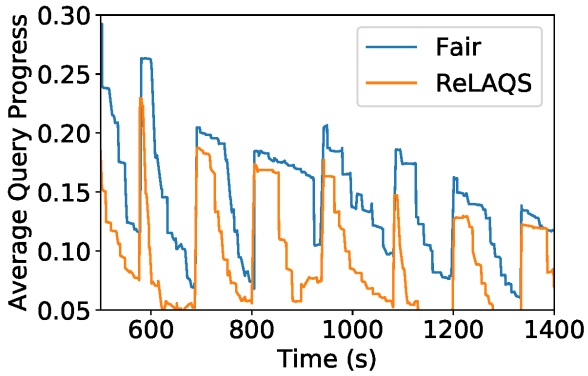
6.1 Methodology

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.

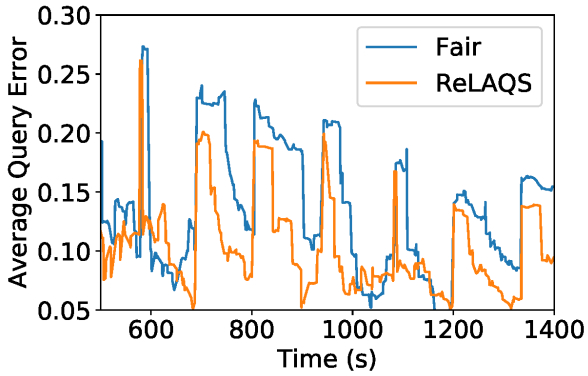
Workload We tested our system with a subset of queries known to be supported by iOLAP[34]. This is a subset of the TPC-H benchmark (queries 1, 3, 5, 6, 11, 16, and 19). In order to diversify the set of queries, we created datasets of varying sizes up to 1 terabyte.

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.

6.2 Simultaneous Query Submission



(a) Average Query Progress



(b) Average Query Normalized Error

Figure 7: 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.

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.

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, 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 7a.

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 7b, 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. Importantly, this metric requires knowledge of a query's true result, so ReLAQS cannot optimize for this metric as it does with the first metric.

Figure 8, 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 8c, 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 8c, 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.

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 8a and 8b 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%.

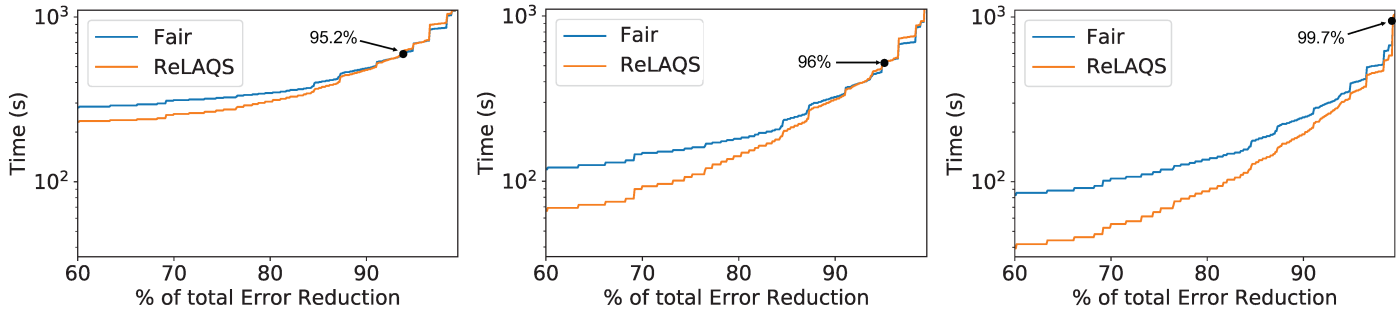
Mean time to arrival's effect on performance Though we have chosen 120 seconds as the mean time to arrival in these experiments, it is also important to understand how the frequency with which queries are submitted affects ReLAQS. In Figure 9, we explore how long it takes the average query to reach 90% 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 are 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.

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

In Figure 10, we take a set of TPC-H queries and use our curve-fitting techniques discussed in section 4. 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



(a) Mean arrival interval=10 seconds (b) Mean arrival interval=40 seconds (c) Mean arrival interval=120 seconds

Figure 8: 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.

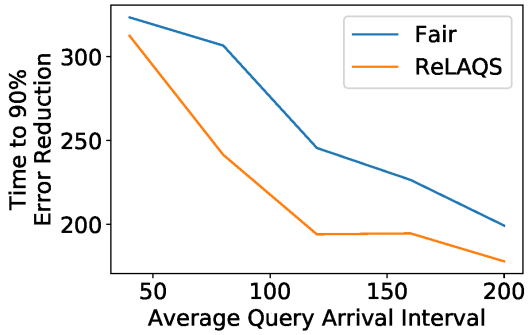


Figure 9: The average time a query takes to reach 90% error reduction with varying arrival frequencies.

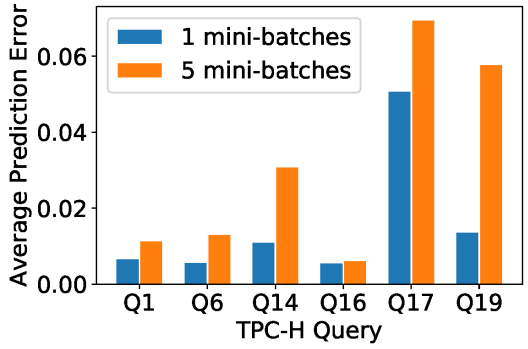


Figure 10: Several TPC-H queries' prediction errors. Average prediction for even noisy queries caps at 5% for 1 mini-batch in advance and 7% for 5 mini-batches in advance.

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.

7. 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.

Investigating more types of queries ReLAQS was built on top of iOLAP [34], another research project that converts standard SparkSQL queries into online aggregation queries. While it allowed us to build ReLAQS on Spark, and largely ignore the implementation of online aggregation, it also presented a set of limitations. In particular, some types of queries were not supported in either Spark SQL 1.4 or iOLAP. Coupled with some non-compact memory representations in Spark SQL, we were limited to a relatively small set of queries in the TPC-H benchmark. In future work, we would like to expand to test a bigger set of queries, but we made sure our subset included queries with uncertain subqueries.

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.

Note that ReLAQS requires no information from the user except the approximate result after each iteration. If a user simply wants to stop a query after an accuracy cutoff point, this is trivial. However, we've found users' accuracy goals are ofthen not this simple, and providing them to ReLAQS would decrease usability.

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 [29]. 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 paper focuses on iterative AQP queries, we are excited about the possibility of having ReLAQS work on a cluster sharing different types of iterative applications, such as machine learning training tasks. If ReLAQS can support different applications together on the same cluster, the potential for increasing cluster utilization grows enormously.

8. Related Work

Approximate Query Processing systems Many systems [4, 6, 8, 15, 18, 28, 33] allow users to get approximate results with significantly reduced job completion time. Online aggregation databases [5, 15, 34] 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. A few systems focus on multi-tenant AQP queries, but they focus largely on application-specific deduplication which is orthogonal to this work [30]. In fact, some focus on the partitioning of online aggregation applications on MapReduce, and the implications of this partitioning on scheduling, but still make the assumption that these applications should each receive an even partitioning of resources [24].

Cluster scheduling systems Existing cluster schedulers [9, 11, 14, 16, 17, 31] 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 tradeoff, 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 [10] or closed-form solutions [21]. Further work to estimate these methods' accuracies [3] 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 [27], as well as the comparative utilities of different visualizations [25]. They have also created loss functions meant to maximize graph accuracy with minimal sampling [26]. These loss functions can be used to map a graph's visualization to progress.

Utility scheduling Utility functions have been widely studied in network traffic scheduling to encode the benefit of performance to users [20, 22, 23]. Recent work on live video analytics [35] leverages utility-based scheduling to provide a universal performance measurement to account for both quality and lag. In addition, recent work on scheduling machine learning applications leverages utility-based scheduling to reduce lag [36].

9. 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.

Acknowledgments

We are grateful to Wyatt Lloyd, David Liu, Robert Macdavid, and Theano Stavrinos for reading early versions of this work and providing feedback. We also thank our shepherd Ioana Giurgiu and all of the anonymous Middleware reviewers for their constructive comments. This work is supported by NSF awards IIS-1250990 and CNS-0953197.

References

[1] Databricks. URL: `http://databricks.com/`.

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

[3] S. Agarwal, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. 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.

[4] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data. In *ACM EuroSys*, 2013.

[5] Y. Ahmad, O. Kennedy, C. Koch, and M. Nikolic. Dbtoaster: Higher-order delta processing for dynamic, frequently fresh views. *Proceedings of the VLDB Endowment*, 5(10):968–979, 2012.

[6] G. Ananthanarayanan, M. C.-C. Hung, X. Ren, I. Stoica, A. Wierman, and M. Yu. GRASS: Trimming Stragglers in Approximation Analytics. In *USENIX NSDI*, 2014.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. 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.

[8] B. Babcock, S. Chaudhuri, and G. Das. Dynamic Sample Selection for Approximate Query Processing. In *ACM SIGMOD*, 2003.

[9] A. A. Bhattacharya, D. Culler, E. Friedman, A. Ghodsi, S. Shenker, and I. Stoica. Hierarchical Scheduling for Diverse Datacenter Workloads. In *ACM SoCC*, 2013.

[10] B. Efron and R. J. Tibshirani. *An introduction to the bootstrap*. CRC press, 1994.

[11] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant Resource Fairness: Fair Allocation of Multiple Resource Types. In *USENIX NSDI*, 2011.

[12] I. Goiri, R. Bianchini, S. Nagarakatte, and T. D. Nguyen. Approxhadoop: Bringing approximations to mapreduce frameworks. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 383–397. ACM, 2015.

[13] M. Habib, C. McDiarmid, J. Ramirez-Alfonsin, and B. Reed. *Probabilistic methods for algorithmic discrete mathematics*, volume 16. Springer Science & Business Media, 2013.

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

[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online Aggregation. In *ACM SIGMOD*, 1997.

[16] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A Platform for Fine-grained Resource Sharing in the Data Center. In *USENIX NSDI*, 2011.

[17] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *ACM SOSP*, 2009.

[18] C. Jermaine, S. Arumugam, A. Pol, and A. Dobra. Scalable Approximate Query Processing with the DBO Engine. *ACM Transactions on Database Systems*, 33(4):23, 2008.

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

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

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

[22] 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.

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

[24] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *Proceedings of the VLDB Endowment*, 4(11), 2011.

[25] A. Parameswaran, N. Polyzotis, and H. Garcia-Molina. Seedb: Visualizing database queries efficiently. *Proc. VLDB Endow.*, 7(4):325–328, Dec. 2013.

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

[27] S. Rahman, M. Aliakbarpour, H. K. Kong, E. Blais, K. Karahalios, A. Parameswaran, and R. Rubinfield. I've seen "enough": Incrementally improving visualizations to support rapid decision making. *Proc. VLDB Endow.*, 10(11):1262–1273, Aug. 2017.

[28] S. Venkataraman, A. Panda, G. Ananthanarayanan, M. J. Franklin, and I. Stoica. The Power of Choice in Data-aware Cluster Scheduling. In *USENIX OSDI*, 2014.

[29] E. Wu, L. Jiang, L. Xu, and A. Nandi. Graphical perception in animated bar charts. *arXiv preprint arXiv:1604.00080*, 2016.

[30] S. Wu, B. C. Ooi, and K.-L. 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.

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

[32] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing. In *USENIX NSDI*, 2012.

[33] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. 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.

[34] K. Zeng, S. Agarwal, and I. Stoica. iOLAP: Managing Uncertainty for Efficient Incremental OLAP. In *ACM SIGMOD*, 2016.

[35] H. Zhang, G. Ananthanarayanan, P. Bodik, M. Philipose, P. Bahl, and M. J. Freedman. Live Video Analytics at Scale with Approximation and Delay-Tolerance. In *USENIX NSDI*, 2017.

[36] H. Zhang, L. Stafman, A. Or, and M. 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.