Knowing When You're Wrong: Building Fast and Reliable Approximate Query Processing Systems

Sameer Agarwal[†], Henry Milner[†], Ariel Kleiner[†], Ameet Talwalkar[†], Michael Jordan[†], Samuel Madden[‡], Barzan Mozafari[*], Ion Stoica[†]

[†]University of California, Berkeley [‡]Massachusetts Institute of Technology [*]University of Michigan, Ann Arbor
[†]{sameerag, henrym, akleiner, ameet, jordan, istoica}@cs.berkeley.edu, [‡]madden@csail.mit.edu, [*]mozafari@umich.edu

ABSTRACT

Modern data analytics applications typically process massive amounts of data on clusters of tens, hundreds, or thousands of machines to support near-real-time decisions. The quantity of data and limitations of disk and memory bandwidth often make it infeasible to deliver answers at interactive speeds. However, it has been widely observed that many applications can tolerate some degree of inaccuracy. This is especially true for *exploratory queries* on data, where users are satisfied with "close-enough" answers if the answer can come quickly. A popular technique for speeding up queries at the cost of accuracy is to execute each query on a sample of data, rather than the whole dataset. To ensure that the returned result is not too inaccurate, past work on approximate query processing has used statistical techniques to estimate "error bars" on returned results. However, existing work in the sampling-based approximate query processing (S-AQP) community has not validated whether these techniques actually generate accurate error bars for real query workloads. In fact, we find that error bar estimation often fails on real world production workloads. Fortunately, it is possible to quickly and accurately *diagnose* the failure of error estimation for a query. In this paper we show that it is possible to implement a query approximation pipeline that produces approximate answers and reliable error bars at interactive speeds.

Categories and Subject Descriptors

H.2.4 [**Systems**]: Query processing

Keywords

Approximate Query Processing; Error Estimation; Diagnostics

1. INTRODUCTION

Sampling-based approximate query processing (S-AQP) has a long history in databases. Nearly three decades ago, Olken and Rotem [27] introduced random sampling in relational databases as a means to return approximate answers and reduce query response times. A large body of work has subsequently proposed different sampling techniques [7, 8, 13, 14, 15, 21, 23, 30, 31]. All of this work shares the same motivation: sampling can dramatically improve the latency and resource costs of queries. Indeed, sampling can produce a *greater-than-linear* speedup in query response time if the sample can fit into memory but the whole dataset cannot, enabling a system to return answers in only a few seconds. Research on human-computer interactions, famously the work of Miller [25], has shown that such quick response times can make a qualitative difference in user interactions.

However, approximate answers are most useful when accompanied by accuracy guarantees. Therefore, a key aspect of almost all S-AQP systems is their ability to *estimate* the *error* of their returned results. Most commonly, error estimates come in the form of confidence intervals (a.k.a. "error bars") that provide bounds on the error caused by sampling. Such error estimates allow the AQP system to check whether its sampling method produces results of reasonable accuracy. They can also be reported directly to users, who can factor the uncertainty of the query results into their analyses and decisions. Further, error estimates help the system control error: by varying the sample size while estimating the magnitude of the resulting error bars, the system can make a smooth and controlled trade-off between accuracy and query time. For these reasons, many methods have been proposed for producing reliable error bars—the earliest being closed-form estimates based on either the central limit theorem (CLT) [32] or on large deviation inequalities such as Hoeffding bounds [19]. Unfortunately, deriving closed forms is often a manual, analytical process. As a result, closed-form-based S-AQP systems [7, 8, 13, 14, 15, 21, 31] are restricted to very simple SQL queries (often with only a single layer of basic aggregates like `AVG`, `SUM`, `COUNT`, `VARIANCE` and `STDEV` with projections, filters, and a group by). This has motivated the use of resampling methods like the bootstrap [23, 30], which require no such detailed analysis and can be applied to arbitrarily complex SQL queries. In exchange, the bootstrap adds some additional overhead.

Beyond these computational considerations, it is critical that the produced error bars be *reliable*, *i.e.,* that they not under- or overestimate the actual error. Underestimating the error misleads users with a false confidence in the approximate result, which can propagate to their subsequent decisions. Overestimating the error is also undesirable. An overestimate of error forces the S-AQP system to use an unnecessarily large sample, even though it could achieve a given level of accuracy using a much smaller sample, and hence much less computation. Approximation is most attractive when it is achieved with considerably less effort than needed for fully accurate results, so inflating sample sizes is problematic.

Unfortunately, no existing error estimation technique is ideal. Large deviation inequalities (*e.g.,* Hoeffding bounds used in [7, 21]) can provide very loose bounds in practice [21], leading to overestimation of error and thus an unnecessary increase in computation.

SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.

ACM 978-1-4503-2376-5/14/06 ...$15.00.
http://dx.doi.org/10.1145/2588555.2593667.

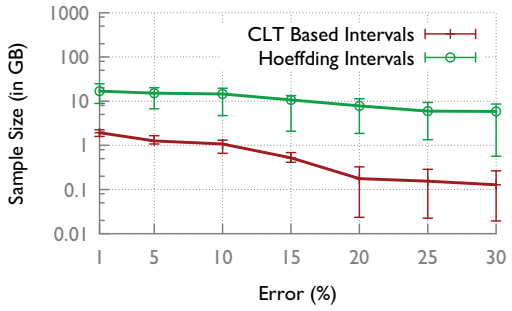


Figure 1: Sample sizes suggested by different error estimation techniques for achieving different levels of relative error.

For instance, in Fig. 1 we show the sample sizes needed to achieve different levels of relative error (averaged over 100 Hive queries from production clusters at Conviva Inc. on tens of terabytes of data[1] with vertical bars on each point denoting .01 and .99 quantiles). If the AQP system were to believe the error estimates of these different techniques, a system relying on Hoeffding bounds must use samples that are 1–2 orders of magnitude larger than what is otherwise needed, diminishing the performance benefits of approximation. On the other hand, CLT-based and bootstrap-based error estimation methods—while being significantly more practical than large deviation inequalities—do not always yield accurate error estimates either. These techniques are guaranteed to work well only under regularity assumptions (in particular, assumptions on the *smoothness* of query aggregates) that are sometimes unrealistic and can produce both underestimates and overestimates in practice. Prior work has not vetted their accuracy.

In fact, as we show in this paper, accuracy is a surprisingly serious problem in practice. When we evaluated these schemes on real queries and data from Facebook (see §3 for details), we found that the most accurate and general estimation technique, the bootstrap, often produces error bars that are far too wide or far too narrow for 23.94% and 12.2% of queries respectively. While CLT-based closed-form techniques are applicable to only 56.78% of the overall queries at Facebook, they also produce incorrect error estimates for 24.86% of the total queries. Though error estimation fails predictably on some kinds of queries on certain kinds of datasets, failure is not easy to predict in most cases.

In AQP, unlike some applications of statistics, it is always possible to fall back to a slower, more accurate solution. In most cases, correct error bars can be computed for any approximate query, though at some expense, by re-executing the query many times over new samples. Of course, the query can also be executed on the full data, obviating the need for error estimation. There is a spectrum of techniques in between these expensive-but-reliable methods and cheap-but-unreliable estimation techniques like closed-forms. What is really missing is a way to quickly identify whether a particular technique will work well for a particular query. With such a tool in hand, we could use cheap error estimation in the common case when it works, while falling back to more expensive methods when the cheap ones fail.

The past decade has seen several investigations of such *diagnostic* methods in the statistics literature [12, 22]. Recently, Kleiner et al. [22] designed a diagnostic algorithm to detect when bootstrap-based error estimates are unreliable. However, their algorithm was not designed with computational considerations in mind and involves tens of thousands of test query executions, making it prohibitively slow in practice.

In this paper, we extend this diagnostic algorithm to validate multiple procedures for generating error bars at runtime and present a series of optimization techniques across all stages of the query processing pipeline that reduce the running time of the diagnostic algorithm from hundreds of seconds to only a couple of seconds, making it a practical tool in a distributed AQP system. As a demonstration of the utility of the diagnostic, we integrate bootstrap based error estimation techniques and the diagnostics into BlinkDB [8], a recent open-sourced AQP database system. With the diagnostic in place, we demonstrate that BlinkDB can answer a range of *complex* analytic queries on large samples of data (of gigabytes in size) at *interactive speeds* (*i.e.,* within a couple of seconds), while falling back to non-approximate methods to answer queries whose errors cannot be accurately estimated. Overall, our contributions are as follows:

1. First, we demonstrate, using a large set of real queries and data at Facebook, that several existing techniques for producing error bars are sometimes dangerously inaccurate. To aid with intuition, we provide some accessible background on how these techniques work, how to evaluate them, and why they can fail.
2. Second, we show that we can use a diagnostic technique to validate multiple procedures for generating error bars at runtime. This makes it possible to diagnose and avoid exposing inaccurate error bars to users in most cases.
3. Third, we provide several optimizations that make the diagnostic and the procedures that generate error bars practical, ensuring that these procedures do not affect the interactivity of the overall query.
4. Finally, with these optimizations, and leveraging recent systems for low-latency exact query processing, we demonstrate a viable end-to-end system for approximate query processing using sampling. We show that this system can deliver interactive-speed results for a wide variety of analytic queries from real world production clusters.

2. BACKGROUND

A key aspect of approximate query processing is estimating the error of the approximation, a.k.a. "error bars". However, existing techniques for computing error bars are often inaccurate when applied to real-world queries and datasets. This result is simply stated, but the intuition behind it may be unclear. We first provide some background on how the large variety of available error estimation techniques work, and why and when they might fail.

2.1 Approximate Query Processing (AQP)

We begin with a brief overview of a query approximation framework. Let θ be the query we would like to compute on a dataset D, so that our desired query answer is $\theta(D)$. For example, consider the following query which returns the average session times of users of an online service from New York City:

```
SELECT AVG(Time)
FROM Sessions
WHERE City = 'NYC'
```

The θ corresponding to this query is:

$$\theta(D) = \frac{1}{N} \sum_{t \in \sigma_{\text{City = 'NYC'}}(D)} t.Time \quad (1)$$

[1]A more detailed description of our datasets and experiments is presented in §7.

where N is the total number of tuples processed and $t.Time$ refers to the $Time$ attribute in tuple t.

It is common for an analytical (a.k.a. OLAP) query to evaluate one or more such aggregate functions, each of which may output a set of values (due to the presence of `GROUP BY`). However, for the sake of simplicity, we assume that each query evaluates a single aggregate function that returns a single real number. When a query in our experimental dataset produces multiple results, we treat each result as a separate query. Additionally note that, while we focus on θs that encode analytical SQL queries, θ could instead correspond to a different aggregation function, like a MapReduce job in EARL [23].

When the size of the dataset ($|D|$) is too large or when the user prefers lower latency than the system can deliver, it is possible to save on I/O and computation by processing the query on less data, resorting to *approximation*. In particular, sampling has served as one of the most common and generic approaches to approximation of analytical queries [7, 8, 13, 14, 15, 21, 23, 30, 31]. The simplest form of sampling is *simple random sampling* with *plug-in estimation*. Instead of computing $\theta(D)$, this method identifies a random sample $S \subseteq D$ by sampling $n = |S| \leq |D|$ rows uniformly at random from D with replacement[2], and then returns the *sample estimate* $\theta(S)$ to the user[3]. Since $\theta(S)$ depends on the particular sample we identified, it is random, and we say that it is a draw from its *sampling distribution* $\text{Dist}(\theta(S))$. The random quantity $\varepsilon = \theta(S) - \theta(D)$ is the *sampling error*; it also has a probability distribution, which we denote by $\text{Dist}(\varepsilon)$.

2.2 Error Estimation

As we have noted, it is critical for any AQP system to know some kind of summary of the typical values of ε for any query. Like $\theta(D)$, this summary must be estimated from S, and this estimation procedure may suffer from error. The particular summary chosen in most AQP systems is the *confidence interval*, a natural way to compactly summarize an error distribution. A procedure is said to generate confidence intervals with a specified *coverage* $\alpha \in [0, 1]$ if, on a proportion exactly α of the possible samples S, the procedure generates an interval that includes $\theta(D)$. Typically, α is set close to 1 so that the user can be fairly certain that $\theta(D)$ lies somewhere in the confidence interval. Thus, confidence intervals offer a guarantee that is attractive to users.

Procedures for generating confidence intervals do not always work well, and we need a way to evaluate them. Unfortunately, to say that a procedure has correct coverage is not enough to pin down its usefulness. For example, a procedure can trivially achieve α coverage by returning $(-\infty, \infty)$ α of the time, and $\varnothing$ the rest of the time; obviously this procedure is not helpful in estimating error.

To resolve this technical problem, we choose to use *symmetric centered confidence intervals*. These do not have the unreasonable behavior of the above example, and further it is possible to evaluate them as estimates of a ground truth value: the interval centered around $\theta(D)$ that covers exactly the proportion α of the sampling distribution of $\theta(S)$, $\text{Dist}(\theta(S))$[4]. Though it is not really a confidence interval, since it is deterministic and always covers $\theta(D)$, in a slight abuse of terminology we call this ground truth value the *true confidence interval*. The procedure for generating symmetric centered confidence intervals from samples closely mimics the definition of the true value. Given a sample, we can estimate $\theta(D)$ as usual by $\theta(S)$. As we detail in the next section, various error estimation techniques can be used to estimate the sampling distribution $\text{Dist}(\theta(S))$. These two estimates ($\theta(S)$ and the estimate of its distribution) are in turn used to produce a confidence interval by finding an interval centered on $\theta(S)$ covering α of the estimated sampling distribution. This interval can be computed by finding the number a that satisfies $P([\theta(S) - a, \theta(S) + a]) = \alpha$, where P is the estimate of $\text{Dist}(\theta(S))$. We can evaluate such a confidence interval by merely comparing its width with that of the true confidence interval, i.e. by computing $\delta = \frac{(\text{true confidence interval width}) - (\text{estimated interval width})}{(\text{true confidence interval width})}$. If this quantity is much above or below zero for a particular query, we can declare that confidence interval estimation has failed in that case. Unlike coverage, this evaluation fully captures problems with error estimation. See [20] for a more detailed discussion of symmetric centered confidence intervals.

2.3 Estimating the Sampling Distribution

We now turn to the problem of estimating $\text{Dist}(\theta(S))$ using only a single sample S. There are three widely-used methods: the bootstrap [16, 39]; closed-form estimates based on normal approximations; and large deviation bounds [24]. It may be unclear why they could fail, and in order to aid intuition we provide a brief explanation of the methods.

2.3.1 Nonparametric Bootstrap

Given unlimited time and access to the entire dataset D, we could compute the sampling distribution to arbitrary accuracy by taking a large number K of independent random samples from D (in exactly the same way as we computed S) and computing θ on each one, producing a distribution over sample estimates. As $K \to \infty$, this would exactly match the sampling distribution.

However, actually accessing D many times would defeat the purpose of sampling, namely, the need to use only a small amount of data. Instead, Efron's *nonparametric bootstrap* [16] (or simply the *bootstrap*) uses the sample S in place of the dataset D, just as we used $\theta(S)$ as an estimate for $\theta(D)$ earlier. This means we take K "resamples" of size n (with replacement[5]) from S, which we denote S^i; compute $\theta(S^i)$ for each one; and take this "bootstrap resampling distribution" as our estimate of the sampling distribution. We make a further approximation by taking K to be merely a reasonably large number, like 100 (K can be tuned automatically [17]).

The substitution of S for D in the bootstrap does not always work well. Roughly, its accuracy depends on two things[6]:

1. **The sensitivity of θ to outliers in D**: If θ is sensitive to rare values and D contains such values, then the estimate can be poor (consider, for example, θ = `MAX`).

2. **The sample size n**: When the sensitivity condition is met, theory only tells us that the estimate is accurate in the limit as $n \to \infty$. The estimate can be poor even for moderately large n (for example 1, 000, 000), and the theory gives no guidance

[2] We assume that samples are taken with replacement only to simplify our subsequent discussion. In practice, sampling *without* replacement gives slightly more accurate sample estimates.

[3] Another function $\hat{\theta}(S)$ may be used to estimate $\theta(D)$. For example, when θ is `SUM`, a reasonable choice of $\hat{\theta}$ would be the sample sum multiplied by a scaling factor $\frac{|D|}{|S|}$. The proper choice of $\hat{\theta}$ for a given θ is not the focus of this paper and has been discussed in [26]. For simplicity of presentation we assume here that θ is given and that it appropriately handles scaling.

[4] This interval is not technically unique, but for moderately-sized D it is close to unique.

[5] Note that, since the resamples are taken with replacement, the S^i are *not* simply identical to S, even though they are all samples of size $n = |S|$ from S.

[6] There are two standard books by Van der Vaart ([34, 35]) that treat this in great detail.

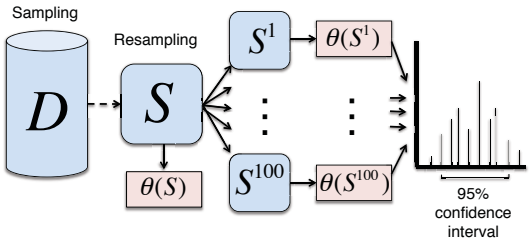


Figure 2: The computational pattern of the bootstrap.

on the requirements for n. It should be noted that the relationship between n and the accuracy of error bars is an exact analogue of the relationship between n and the size of the sampling error $|\varepsilon|$. It is therefore unlikely that the dependence on n will be avoided by finding a different method for computing confidence intervals; any reasonable method will suffer from some inaccuracy at small n and improve in accuracy as n grows.

In addition to the possibility of failure, the bootstrap involves a large amount of computation (K replications of the query on resamples), which can partially offset the benefits of sampling. Fig. 2 illustrates the use of the bootstrap in computing confidence intervals and its potential cost. This motivates the consideration of alternative methods.

2.3.2 *Normal approximation and closed-form estimate of variance*

This method approximates the sampling distribution by a normal distribution $N(\theta(S), \sigma^2)$ and estimates the variance σ^2 by a special closed-form function of the sample. We call this *closed-form estimation* for short. The normal approximation is justified by appealing to the central limit theorem. Like the bootstrap, this rests on assumptions of insensitivity of θ to outliers in D and on n being large. However, closed-form estimation replaces the brute-force computation of the bootstrap with estimation of the parameter σ^2 from S through careful manual study of θ. For example, a well-known estimate for σ^2 when $\theta(S)$ is `AVG` is $s^2 = \frac{\mathrm{Var}(S)}{n}$. Calculating $\mathrm{Var}(S)$ is much cheaper than computing K bootstrap replicates, so in the case of `AVG`, closed-form estimation is appealing. The story is similar for several θs commonly used in SQL queries: `COUNT`, `SUM`, `AVG`, and `VARIANCE`. Other θs require more complicated estimates of σ^2, and in some cases, like `MIN`, `MAX`, and black-box *user defined functions* (UDFs), closed-form estimates are unknown. Therefore closed-form estimation is less general than the bootstrap. In our Facebook trace, 37.21% of queries are amenable to closed-form estimates. Fig. 7(a) shows the overhead of estimating the error using closed forms for a set of 100 randomly chosen queries that computed `COUNT`, `SUM`, `AVG`, or `VARIANCE` from our Facebook trace. When closed-form estimation is possible, it generally runs faster than the bootstrap.

2.3.3 *Large Deviation Bounds*

Large deviation bounds[7] [24] are a third technique for error estimation, used in several existing AQP systems including OLA[21, 19] and Aqua[7]. Rather than directly estimating the sampling distribution like the bootstrap and closed forms, this method bounds the *tails* of the sampling distribution; these bounds are sufficient to compute confidence intervals. The method relies on the direct computation of a quantity related to the "sensitivity to outliers" (for example, $|\max D - \min D|$ in the case of `SUM`) that, as previously mentioned, can ruin the accuracy of bootstrap- and closed-form-based error estimation. This sensitivity quantity depends on D and θ, so it must be precomputed for every θ and, like σ^2 in closed-form estimation, requires difficult manual analysis of θ. By making a worst-case assumption about the presence of outliers, large deviation bounds ensure that the resulting confidence intervals never have coverage less than α. That is, error bars based on large deviation bounds users will never be too small. However, this conservatism comes at a great cost: as we observe in Fig. 1, typically large deviation bounds instead produce confidence intervals much wider than the true confidence interval and with coverage much higher than α, making them extremely inefficient in practice.

[7] There are many examples of such bounds, including Hoeffding's inequality, Chernoff's inequality, Bernstein's inequality, McDiarmid's inequality, or the bounded-differences inequality.

3. PROBLEM: ESTIMATION FAILS

As noted in the previous section, all three estimation methods have modes of failure. We've already highlighted the pessimistic nature of large deviation bounds in Fig. 1, and in this section we evaluate the severity of these failure modes for the bootstrap and closed-form estimation. Recall that we can summarize the accuracy of a symmetric centered confidence interval by δ, the relative deviation of its width from that of the true interval. Ideally, we would like δ to be close to $zero$. If it is often positive and large, this means our procedure produced confidence intervals that are too large, and the intervals are typically larger than the true sampling error ε. In that case, we say that the procedure is *pessimistic*. (In Fig. 1 we saw an example– confidence intervals based on Hoeffding bounds suffer from extreme pessimism.) On the other hand, if δ is often much smaller than 0, then our procedure is producing confidence intervals that are misleadingly small, and we say that it is *optimistic*.

The two cases result in different problems. A pessimistic error estimation procedure will cause the system to use inefficiently large samples. For example, say that for a particular query and sample size we have $\theta(D) = 10$, $\theta(S) = 10.01$, and we produce the confidence interval $[5.01, 15.01]$. If the user requires a relative error no greater than 0.1%, the system is forced to use a much larger sample, even though $\theta(S)$ is actually accurate enough. Without a fixed requirement for error, pessimistic error estimation will lead the user to disregard $\theta(S)$ even when it is a useful estimate of $\theta(D)$.

While pessimistic error estimation results in decisions that are too conservative, an optimistic error estimation procedure is even worse. If for a different query we have $\theta(D) = 10$, $\theta(S) = 15$, and the system produces the confidence interval $[14.9, 15.1]$, then the user will make decisions under the assumption that $\theta(D)$ probably lies in $[14.9, 15.1]$, even though it is far away. Since pessimism and optimism result in qualitatively different problems, we present them as two separate failure cases in our evaluation of error estimation procedures.

To test the usefulness of error estimation techniques on real OLAP queries, we analyzed a representative trace of 69,438 Hive queries from Facebook constituting a week's worth of production queries during the 1^{st} week of Feb 2013 and a trace of 18,321 Hive queries from Conviva Inc. [4] constituting a sample of a month's worth of production queries during Feb 2013. Among these, `MIN`, `COUNT`, `AVG`, `SUM`, and `MAX` were the most popular aggregate functions at Facebook constituting 33.35% , 24.67%, 12.20%, 10.11% and 2.87% of the total queries respectively. 11.01% of these queries consisted of one or more *UDFs* or *User-Defined Functions*. In Conviva on the other hand, `AVG`, `MAX`, `COUNT`, and `PERCENTILES` were the most popular aggregate functions with a combined share of 32.3%. 42.07% of the Conviva queries had at least one UDF. While we are

unable to disclose the exact set of queries that we used in our analysis due to the proprietary nature of the query workload and the underlying datasets, we have published a synthetic benchmark [1] that closely reflects the key characteristics of the Facebook and Conviva workloads presented in this paper– both in terms of the distribution of underlying data and the query workload. The input data set in this benchmark consists of a set of unstructured HTML documents and SQL tables that were generated using Intel's Hadoop benchmark tools [6] and data sampled from the Common Crawl [3] document corpus. The set of aggregation queries and the UDFs in the benchmark are characteristic of the workload used at Conviva.

For each query in our dataset, we compute the true values of $\theta(D)$ and the true confidence interval and then run the query on 100 different samples of fixed size $n = 1,000,000$[8] rows. For each run, we compute a confidence interval using an error estimation technique, from which we get δ. We then pick a reasonable range of $[-0.2, 0.2]$, and if δ is outside this range for at least 5% of the samples, we declare error estimation a failure for the query. We present separately the cases where $\delta > 0.2$ (*i.e., pessimistic* error estimation) and where $\delta < -0.2$ (*i.e., optimistic* and *incorrect* error estimation). In addition, since only queries with `COUNT, SUM, AVG,` and `VARIANCE` aggregates are amenable to closed-form error estimation, while all aggregates are amenable to the bootstrap, 43.21% of Facebook queries and 62.79% Conviva queries can only be approximated using bootstrap based error estimation methods. Results are displayed in Fig. 3.

Queries involving `MAX` and `MIN` are very sensitive to rare large or small values, respectively. In our Facebook dataset, these two functions comprise 2.87% and 33.35% of all queries, respectively. Bootstrap error estimation fails for 86.17% of these queries. Queries involving UDFs, comprising 11.01% of queries at Facebook and 42.07% of queries at Conviva, are another potential area of concern. Bootstrap error estimation failed for 23.19% of these queries.

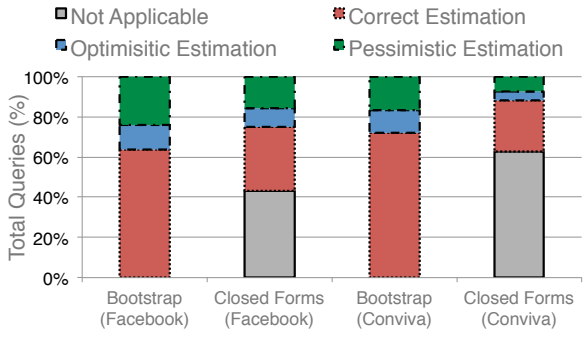


Figure 3: Estimation Accuracy for bootstrap and closed-form based error estimation methods on real world Hive query workloads from Facebook (69, 438 queries) and from Conviva Inc. (18, 321 queries).

4. DIAGNOSIS

It is clear from this evaluation that no type of error estimation gives completely satisfactory results. Lacking a one-size-fits-all error estimation technique, we need an algorithm to identify the cases where the non-conservative methods work and the cases where we must use conservative large deviation bounds or avoid sampling altogether. We call such a procedure a *diagnostic*. We are going to use a diagnostic recently developed by Kleiner et al. [22], but first let us provide some intuition for how a diagnostic should work. We consider first an impractical *ideal* diagnostic.

To check whether a particular kind of error estimation for a query θ on data S is likely to fail, we could simply perform the evaluation procedure we used to present results in the previous section. That is, we could sample repeatedly (say $p = 100$ times) from the underlying dataset D to compute the true confidence interval, estimate a confidence interval on each sample using the error estimation method of interest, compute δ for each one, and check whether most of the δs are close to 0. If we use sufficiently many samples, this ideal procedure will tell us exactly what we want to know. However, it requires repeatedly sampling large datasets from D (and potentially executing queries on K bootstrap resamples for each of these samples), which is prohibitively expensive.

This is reminiscent of the problem of computing the confidence intervals themselves. A simple solution in that case was the bootstrap, which approximates the sampling distribution by resampling from the sample S. We could apply the same idea here by replacing θ with the bootstrap error estimation procedure, thus *bootstrapping* the bootstrap. However, in this case we are trying to test whether the bootstrap itself (or some related error estimation technique) works. If the bootstrap provides poor error estimates, it may also work poorly in this diagnostic procedure. So we need something else.

Instead, we can also approximate the ideal diagnostic by performing it on a sequence of much smaller samples and extrapolating the results to the actual sample, S. This is the basis of the diagnostic procedure of Kleiner et al. It is motivated computationally by the following observation: If S is a simple random sample from D, then subsamples generated by disjointly partitioning S are themselves mutually independent simple random samples from D. Thus by partitioning S we can identify small samples from D without performing additional I/O. Of course, the effectiveness of an error estimation technique on small samples may not be predictive of its effectiveness on S. Therefore careful extrapolation is necessary: we must perform the procedure at a sequence of increasing sample sizes, $(b_1, ..., b_k)$ and check whether the typical value of δ decreases with b_i and is sufficiently small for the largest sample size b_k. The disjointness of the partitions imposes the requirement that each sample size b_i, multiplied by the number of samples p, be smaller than S.

4.1 Kleiner et al.'s Diagnostics

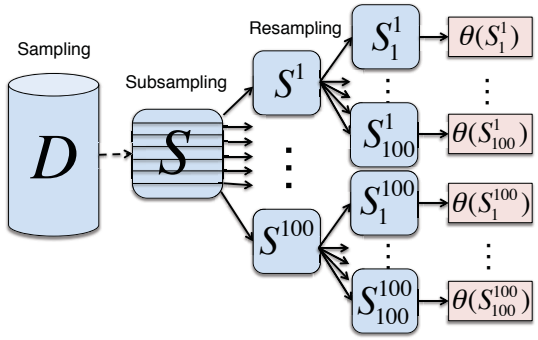
The details of the checks performed by the diagnostic are not critical for the remainder of this paper. The diagnostic algorithm, as applied to query approximation, is described precisely in Algorithm 1 in Appendix A. Fig. 4(a) depicts the pattern of computation performed by the diagnostic.

Kleiner et al. targeted the bootstrap in their work, and carried out an evaluation only for the bootstrap, but also noted in passing that the diagnostic can be applied in principle to any error estimation procedure, including closed-form CLT-based error estimation, simply by plugging in such procedures for ξ. However, this assertion needs evaluation, and indeed it is not immediately clear that the diagnostic will work well for query approximation, even for the bootstrap. The extrapolation procedure is merely an heuristic, with no known guarantees of accuracy.

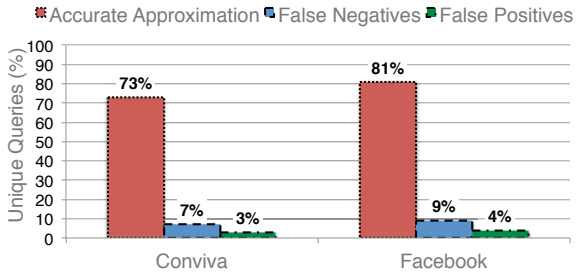
4.2 Diagnosis Accuracy

We demonstrate the diagnostic's utility in query approximation by providing a thorough evaluation of its accuracy on Facebook and Conviva queries for both bootstrap and closed form error estimation. We used two different datasets to evaluate the diagnostic's effectiveness on real world workloads. The first dataset was derived from a subset of 350 Apache Hive [33] queries from Conviva Inc. [4].

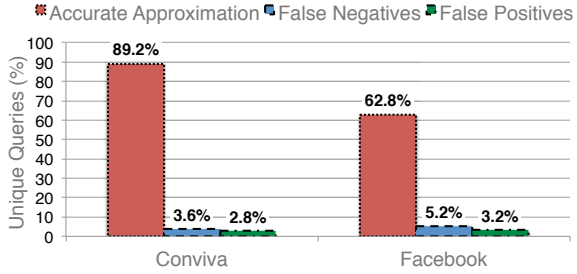
[8] $n = 1,000,000$ was chosen so that the query running time could fairly be called "interactive".



(a) Diagnostic's Computation Diagram



(b) Closed Form Diagnostic Experiments



(c) Bootstrap Diagnostic Experiments

Figure 4: Fig. 4(a) shows the pattern of computation performed by the diagnostic algorithm for a single subsample size. Here the procedure being checked is the bootstrap, so each subsample is resampled many times. For each subsample S^j, the bootstrap distribution $(\theta(S_1^j), \theta(S_2^j), ...)$ is used to compute an estimate of ξ, which is assessed for its closeness to the value of ξ computed on the distribution of values $(\theta(S^1), \theta(S^2), ...)$. The computation pictured here is performed once for each subsample size b_i, using the same sample S. Figures 4(b) and 4(c) compare the diagnostics success for Closed Form and Bootstrap error estimation respectively. For 4(b), we used a workload of 100 queries each from Conviva and Facebook that only computed `AVG`, `COUNT`, `SUM` or `VARIANCE` based aggregates. For 4(c) we used a workload of 250 queries each from Conviva and Facebook that computed a variety of complex aggregates instead.

100 of these queries computed `AVG`, `COUNT`, `SUM` or `VARIANCE`, and the remaining 250 included more complicated expressions for which error estimates could be obtained via the bootstrap. This dataset was 1.7 *TB* in size and consisted of 0.5 billion records of media accesses by Conviva users. The second dataset was one derived from Facebook Inc. and was approximately 97.3 *TB* in size, spanning over 40 billion records in different tables. As with the Conviva queries, we picked sets of 100 and 250 queries for evaluation respectively suited for closed-forms and bootstrap based error estimation

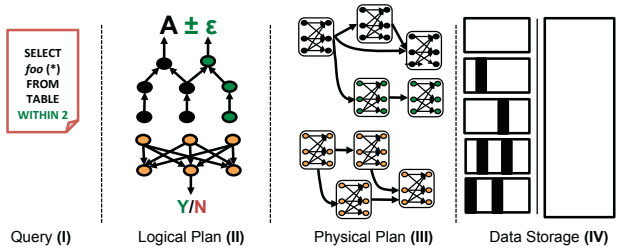


Figure 5: Workflow of a Large-Scale, Distributed Approximate Query Processing Framework

techniques. The performance of the diagnostic algorithm is a function of a variety of statistical parameters all of which affect the accuracy of the underlying diagnosis. Fig. 4 compares the success accuracy of the diagnostics over the set of queries from Facebook and Conviva for our experiments. Overall, with these diagnostic settings in place, we predicted that 84.57% of Conviva queries and 68% of Facebook queries can be accurately approximated while the others cannot, with less than 3.1% false positives and 5.4% false negatives.

5. ERROR ESTIMATION PIPELINE ARCHITECTURE

At this point we have all the necessary ingredients to sketch a simple scheme for query approximation. Fig. 5 describes an end-to-end workflow of a large-scale distributed approximate query processing framework that computes approximate answers, estimates errors and verifies its correctness. The *Query (I)* is first compiled into a *Logical Query Plan (II)*. This in turn has three distinct parts: one that computes the approximate answer "$\theta(S)$", another that computes the error "$\hat{\xi}$" and finally the component that computes the diagnostic tests. Each logical operator is instantiated by the *Physical Query Plan (III)* as a DAG of tasks. These tasks execute and communicate in parallel, and operate on a set of samples that are distributed across multiple disks or cached in the memory. Finally, the *Data Storage Layer (IV)* is responsible for efficiently distributing these samples across machines and deciding which of these samples to cache in memory. Notice that there are several steps that are necessary in computing queries on sampled data. If we are to fulfill our promise of interactive queries, each of these steps must be fast. In particular, hundreds of bootstrap queries and tens of thousands of small diagnostic queries must be performed within seconds. In this section, we will first describe the *Poissonized Resampling* technique (§5.1) that enables us to efficiently create multiple resamples by simply streaming the tuples of the original sample, then present a baseline solution based on this technique (§5.2), and finally propose a number of query plan optimizations (§5.3) to achieve interactivity.

5.1 Poissonized Resampling

Recall that the bootstrap (including the bootstraps performed on the small subsamples used in the diagnostic) requires the identification of many *resampled* datasets from a given sample S. To compute a resample from S, we take $|S|$ rows with replacement from it. Equivalently, we can also view resampling as assigning a random count in $\{0, 1, ..., |S|\}$ to each row of S according to a certain distribution, with the added constraint that the sum of counts assigned to all rows is exactly $|S|$. The distribution of counts for each row is `Poisson(1)`; this distribution has mean 1, and counts can be sampled from it quickly [28]. However, the sum constraint of having exactly $|S|$ tuples couples the distribution of row counts and substantially complicates the computation of resamples, especially

when samples are partitioned on multiple machines. The coupled resample counts must be computed from a large multinomial distribution and stored, using $O(|S|)$ memory per resample. Pol and Jermaine showed that a Tuple Augmentation (TA) algorithm designed to sample under this constraint not only incurred substantial preprocessing overheads but was also in general $8-9\times$ slower than the non-bootstrapped query [30].

However, statistical theory suggests that the bootstrap does not actually require that resamples contain exactly $|S|$ elements, and the constraint can simply be eliminated. The resulting approximate resampling algorithm simply assigns independent counts to each row of S, drawn from a `Poisson(1)` distribution. It can be shown that this "Poissonized" resampling algorithm is equivalent to ordinary resampling except for a small random error in the size of the resample. Poissonized resamples will contain $\sum_{i=1}^{|S|}$`Poisson(1)` elements, which is very close to $|S|$ with high probability for moderately large $|S|$. For example, if $|S| = 10,000$, then $P(\text{Poissonized resample size} \in [9500, 10500]) \cong 0.9999994$; generally the Poissonized resample count is approximately $Normal(\mu = |S|, \sigma = \sqrt{|S|})$. Please see Chapter 3.7 of [35] for a discussion of Poissonization and the bootstrap. Creating resamples using Poissonized resampling is embarrassingly parallel, requires no extra memory if each tuple is immediately pipelined to downstream operators, and is extremely fast. This forms the basis of our implementation of the bootstrap and diagnostics.

5.2 Baseline Solution

A simple way to implement our error estimation pipeline is to add support for the Poissonized Resampling operator which can be invoked in SQL as "`TABLESAMPLE POISSONIZED (100)`". As the data is streamed through the operator, it simply assigns independent integral weights to each row of S, drawn from a `Poisson(1)` distribution. (The number in parentheses in the SQL is the rate parameter for the Poisson distribution, multiplied by 100.) One weight is computed for each row for every resample in which the row potentially participates. With support for a Poissonized resampling operator in place, the bootstrap error approximation for a query can be straightforwardly implemented by a simple SQL rewrite rule. For instance, bootstrap error on the sample table "`S`" for a simple query of the form "`SELECT foo(col_S) FROM S`" can be estimated by rewriting the query as a combination of a variety of subqueries (each of which computes an answer on a resample) as follows:

```
SELECT foo(col_S), ξ̂(resample_answer) AS error
FROM (
       SELECT foo(col_S) AS resample_answer
       FROM S TABLESAMPLE POISSONIZED (100)
         UNION ALL
       SELECT foo(col_S) AS resample_answer
       FROM S TABLESAMPLE POISSONIZED (100)
         UNION ALL
       ...
         UNION ALL
       SELECT foo(col_S) AS resample_answer
       FROM S TABLESAMPLE POISSONIZED (100)
)
```

Implementing error estimation or the diagnostics (for both bootstrap and closed forms) in the query layer similarly involves either plugging in appropriate error estimation formulas or writing a query to execute subqueries on small samples of data and compare the estimated error with the true error, respectively. With this scheme, the bootstrap requires execution of 100 separate subqueries, and a diagnostic query compiles to 30,000 subqueries (we use $K = 100$ bootstrap resamples, and set $p = 100$ and $k = 3$ in execution of Algorithm 1 settings). Unfortunately, the overhead introduced by executing such a large number of subqueries compromise the interactivity of this approach, even when leveraging distributed in-memory data processing frameworks such as Shark [37] and Spark [38].

There are several overheads incurred by this naive solution. First, both bootstrap and diagnostics are performing the same set of queries over and over again on multiple samples of data. Second, each query further gets compiled into one or more tasks. As the number of tasks grows into thousands, the per-task overhead and the contention caused by each task in continuously resampling the same sample adds up substantially. This suggests that achieving interactivity may require changes to the entire query processing framework.

Prior work, including TA/ODM [30] and EARL [23] have proposed several optimizations to reduce the repetitive work of the bootstrap. While we build on some of these techniques, none of them aims to provide interactive response times. For example, EARL is built to run on top of Hadoop MapReduce, which (at least at the time of that work) was unable to run even small queries interactively due to scheduling overhead and I/O costs. On the other hand, *Tuple Augmentation* (TA) and *On-Demand Materialization* (ODM) based algorithms incur substantial overheads to create exact samples with replacement. Next we will show that leveraging Poissonized resampling techniques to create resamples from the underlying data not only alleviates the need for directly building on these solutions but it is also orders of magnitude more efficient both in terms of runtime and resource usage.

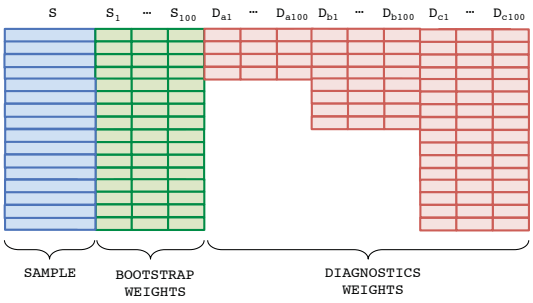
5.3 Query Plan Optimizations

Given the limitations of existing solutions, next we will demonstrate the need for re-engineering the query processing stack (*i.e.,* the logical plan, the physical plan and the storage layer). We start with optimizing the query plan. While some of these optimizations are particularly well suited for parallel approximate query processing frameworks, others are more general in nature.

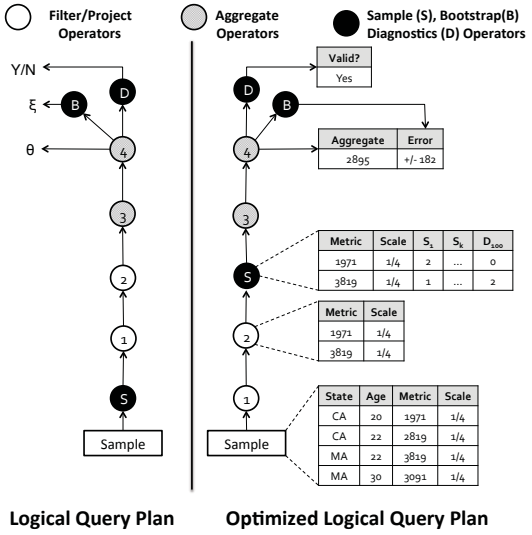
5.3.1 Scan Consolidation

One of the key reasons behind the inefficiency of the bootstrap and the diagnostics tests is the overhead in executing the same query repeatedly on different resamples of the same data. Even if each subquery is executed in parallel, the same input that is streamed through each of these subqueries individually passes through the same sets of filter, projection and other *in-path* operators in the query tree before finally reaching the aggregate operator(s). Furthermore, in a parallel setting, each subquery contends for the same set of resources—the underlying input, the metastore and the scheduler resources, not only making it extremely inefficient but also affecting the overall throughput of the system.

To mitigate these problems, we reduce this problem to one that requires a *single scan* of the original sample to execute all bootstrap sub-queries to estimate the error, and all diagnosis sub-queries to verify the error accuracy. To achieve this goal, we first optimize the logical plan by extending the simple Poissonized Resampling operator in §5.2 to simultaneously augment the tuple with multiple sets of resampling weights– each corresponding to a resample that may either be required to estimate the error using bootstrap or to verify the accuracy of estimation using the diagnostics. More specifically, as shown in Fig. 6(a), to estimate sampling error using bootstrap, for each tuple in sample S, we augment it by associating a set of 100 independent weights S_1, ..., S_{100}, each drawn from a `Poisson(1)` distribution to create 100 resamples of S. For the diagnostics, we



(a) Scan Consolidation



(b) Rewriting the Logical Query Plan

Figure 6: Logical Query Plan Optimizations

first logically partition[9] the sample S into multiple sets of 50 MB, 100 MB and 200 MB and then associate the weights D_{a1}, ..., D_{a100}; D_{b1}, ..., D_{b100} and D_{c1}, ..., D_{c100} to each row in order to create 100 resamples for each of these three sets. Overall we use 100 instances of these three sets for accurate diagnosis. Executing the error estimation and diagnostic queries in a *single pass* further enables us to leverage a lot of past work on efficiently scheduling multiple queries by sharing a single cursor to scan similar sets of data [10, 18, 36].

These techniques also warrant a small number of other straightforward performance optimizations to the database's execution engine by modifying all pre-existing aggregate functions to directly operate on weighted data which alleviates the need for duplicating the tuples before they were streamed into the aggregates. Last but not the least, we also add two more operators to our database's logical query plan– the *bootstrap* operator and the *diagnostic* operator. Both these operators expect a series of point estimates obtained by running the query on multiple resamples of underlying data and estimate the bootstrap error and estimation accuracy respectively.

5.3.2 Operator Pushdown

Ideally, as shown in Fig. 6(b) (left), the *Poissonized resampling operator* should be inserted immediately after the `TABLESCAN` operator in the query graph and the *bootstrap* and *diagnostic* operators should be inserted after the final set of aggregates. This however results in wasting resources on unnecessarily maintaining weights of tuples that may be filtered upstream before reaching the aggregates. Therefore, to further optimize the query plan, we added a *logical plan rewriter* that rewrites the logical query plan during the optimization phase. Rewriting the query plan involves two steps. First, we find the longest set of consecutive *pass-through*[10] operators in the query graph. Second, we insert the custom *Poissonized resampling operator* right before the first non *pass-through* operator in the query graph. This procedure is illustrated in Fig. 6(b) wherein we insert the *Poissonized resampling operator* between stage 2 and 3. The subsequent aggregate operator(s) is(are) modified to compute a set of resample aggregates by appropriately scaling the coresponding aggregation column with the weights associated with every tuple. These set of resample aggregates are then streamed into the *bootstrap* and *diagnostic* operators.

We note that while the *Poissonized resampling operator* temporarily increases the overall amount of intermediate data (by adding multiple columns to maintain resample scale factors), more often than not, the actual data used by the *Poissonized resampling operator* (after the series of filters, projections and scans) is just a tiny fraction of the input sample size. This results in several orders of magnitude reduction of the overall error estimation overhead using the bootstrap.

6. PERFORMANCE TRADEOFFS

We implemented our solution in BlinkDB [8, 9], an open source distributed AQP framework. BlinkDB is built on top of Shark [37], a query processing framework, which in turn is built on top of Spark [38], an execution engine. Shark supports caching inputs and intermediate data in fault tolerant data structures called RDDs (Resilient Distributed Datasets). BlinkDB leverages Spark and Shark to effectively cache samples in memory and to allow users to pose SQL-like aggregation queries (with response time or error constraints) over the underlying data, respectively. Queries over multiple terabytes of data can hence be answered in seconds, accompanied by meaningful error bars bracketing the answer that would be obtained if the query ran instead on the full data. The basic approach taken by BlinkDB is exactly the same as described above—it precomputes and maintains a carefully chosen collection of samples of input data, selects the best sample(s) at runtime for answering each query, and provides meaningful error bounds using statistical sampling theory. We have implemented our solution in BlinkDB version 0.1.1 [2]. Furthermore, we have also helped in successfully integrating the same set of techniques in Facebook's Presto [5], another well-known open-source distributed SQL query engine.

As described in §5, first we added the *Poissonized Sampling* operator, the *bootstrap* operator and the *diagnostics* operator in BlinkDB, then modified the aggregate functions to work on weighted tuples and finally implemented the logical plan rewriter to optimize the query plan for error estimation and diagnostics. While the aforementioned query plan optimizations brought the end to end query latency to tens of seconds, to achieve interactivity, we need a fine grain control on 3 key aspects of the physical plan– the query's degree of parallelism, per-task data locality, and straggler mitigation.

6.1 Degree of Parallelism

BlinkDB maintains a variety of disjoint random samples of the underlying data that is cached across a wide range of machines. These

[9] Please note that the sample S is completely shuffled in the cluster and any subset of S is also a random sample.

[10] We loosely define *pass-through* as those set of operators that do not change the statistical properties of the set of columns that are being finally aggregated. These operators are relatively simple to identify during the query planning and analysis phase [11] and consist of, but are not limited to `scans, filters, projections` etc.

samples are randomly shuffled across the cluster and any subset of a particular sample is a random sample as well. Given that the ability to execute the query on any random sample of data (on any subset of machines) makes the implementation of both the error estimation and diagnostic procedures embarrassingly parallel (except the aggregation step at the end), parallelizing the logical query plan to run on many machines (operating on a fraction of data) significantly speeds up end-to-end response times. However, given the final aggregation step, arbitrarily increasing the parallelism often introduces an extra overhead, even in an interactive query processing framework like BlinkDB. A large number of tasks implies additional per-task overhead costs, increased many-to-one communication overhead during the final aggregation phase, and a higher probability of stragglers. Striking the right balance between the degree of parallelism and the overall system throughput is primarily a property of the underlying cluster configuration and the query workload. We will revisit this trade-off in §7.

6.2 Per-Task Data Locality

While executing a query on a sample, we ensure that a large portion of the samples are cached in cluster memory, using BlinkDB's caching facilities. However we observed empirically that using all the available RAM for caching inputs is not a good idea. In general, given that the total RAM in a cluster is limited, there is a tradeoff between caching a fraction of input data versus caching intermediate data during the query's execution. Caching the input invariably results in faster scan times, but it decreases the amount of per-slot memory that is available to the query during execution. While this is once again a property of the underlying cluster configuration and the query workload, we observed that caching a fixed fraction of samples and allotting a bigger portion of memory for caching intermediate data during the query execution results in the best average query response times. We will revisit this trade-off in §7.

6.3 Straggler Mitigation

With a fine grained control over the degree of parallelism and per-task locality, avoiding straggling tasks during query execution results in further improvements in query runtime. In order to reduce the probability of a handful of straggling tasks slowing down the entire query, we always spawn 10% more tasks on identical random samples of underlying data on a different set of machines and as a result, do not wait for the last 10% tasks to finish. Straggler mitigation does not always result in end-to-end speedups and may even, in theory, introduce bias in error estimation when long tasks are systematically different than short ones [29]. However, in our experiments we observed that straggler mitigation speeds up queries by hundreds of milliseconds and causes no deterioration in the quality of our results.

7. EVALUATION

We evaluated our performance on two different sets of 100 real-world Apache Hive queries from the production clusters at Conviva Inc. [4]. The queries accessed a dataset of size 17 TB stored across 100 Amazon EC2 *m1.large* instances (each with 4 ECUs[11] (EC2 Compute Units), 7.5 GB of RAM, and 840 GB of disk). The cluster was configured to utilize 75 TB of distributed disk storage and 600 GB of distributed RAM cache. The two query sets consists of:

- **Query Set 1** (`QSet-1`): 100 queries for which error bars can be calculated using closed forms (*i.e.,* those with simple `AVG, COUNT, SUM, STDEV` and `VARIANCE` aggregates).
- **Query Set 2** (`QSet-2`): 100 queries queries for which error bars could only be approximated using the bootstrap (*i.e.,* those with multiple aggregate operators, nested subqueries or with *User Defined Functions*).

[11]Each ECU is considered to be equivalent of a 1.0-1.2 GHz Opteron or Xeon processor.

7.1 Baseline Results

Fig. 7(a) and Fig. 7(b) plot the end-to-end response times for executing the query on a sample, estimating the error and running the diagnostics using the naive implementation described in §5.2 on `QSet-1` and `QSet-2` respectively. Every query in these `QSets` is executed with a 10% error bound on a cached random sample of at most 20 GB in size from the underlying 17 TB of data. Given that each of the 3 steps—the query execution on the sample, the error estimation and running the diagnosis—happens in parallel, for each query we plot the *Query Response Time* (*i.e.,* the time it takes to execute the query on a sample), the *Error Estimation Overhead* (*i.e.,* the *additional* overhead of estimating bootstrap or closed form error), and the *Diagnostics Overhead* (*i.e.,* the *additional* overhead of running the diagnosis) separately. These results clearly highlight that simply rewriting the queries to implement error estimation and diagnostics on even relatively small sample sizes typically takes several minutes to run (and more importantly cost 100× to 1000× more resources), not only making it too slow for interactivity but also hugely inefficient. Next, we will revisit our optimizations from Sections §5.3 and §6, and demonstrate their individual speedups in response times with respect to our baseline solution.

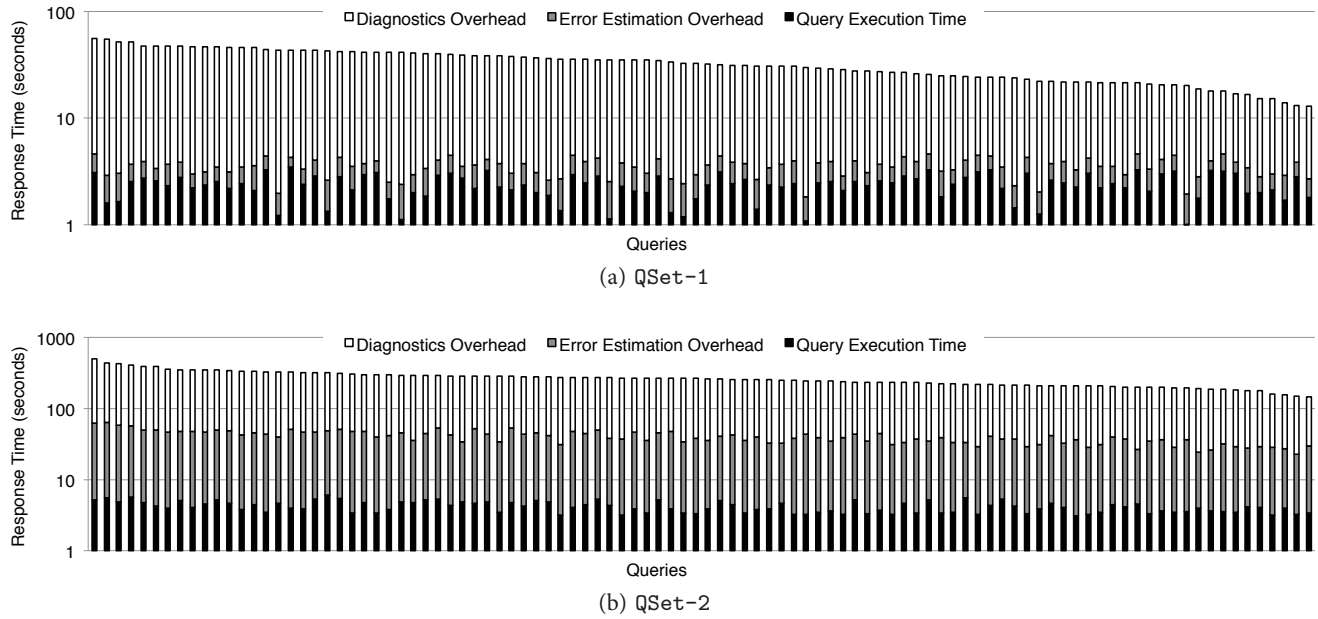
7.2 Query Plan Optimizations

As expected, some of our biggest reductions in end-to-end query response times come from *Scan Consolidation* and *Operator Pushdown*. Fig. 8(a) and Fig. 8(b) show the cumulative distribution function of speedups yielded by query plan optimization techniques (*Scan Consolidation* and *Operator Pushdown*) for error estimation and diagnostics (larger overhead ratios indicate greater speedups). Individually, Fig. 8(a) and Fig. 8(b) show the speedups associated with `QSet-1` & `QSet-2` respectively. These speedups are calculated with respect to the baseline established in Fig. 7(a) and Fig. 7(b) and demonstrate improvements of 1−2× and 5−20× (for error estimation and diagnostics respectively) in `QSet-1`, and 20−60× and 20−100× (for error estimation and diagnostics respectively) in `QSet-2`.

7.3 Performance Tuning

With the query plan optimizations in place, next we will show the benefits of tuning the underlying physical execution plan by varying the degree of parallelism and input cache sizes. As we explain in §6, we observed that striking the right balance between the degree of parallelism and the overall system throughput is primarily a property of the underlying cluster configuration and the query workload. We verified this observation empirically by varying the amount of maximum parallelism for each query. As shown in Fig. 8(c), in our experiments, we observed that both bootstrap based error estimation and the diagnostic procedure were most efficient when executed on up to 20 machines. Increasing the degree of parallelism may actually lead to worse performance as the task scheduling and communication overheads offsets the parallelism gains. Again, note that the optimal degree of parallelism we report here is an artifact of our query load and the sample sizes we picked for our diagnostic procedure. Choosing the degree of parallelism automatically is a topic of future work.

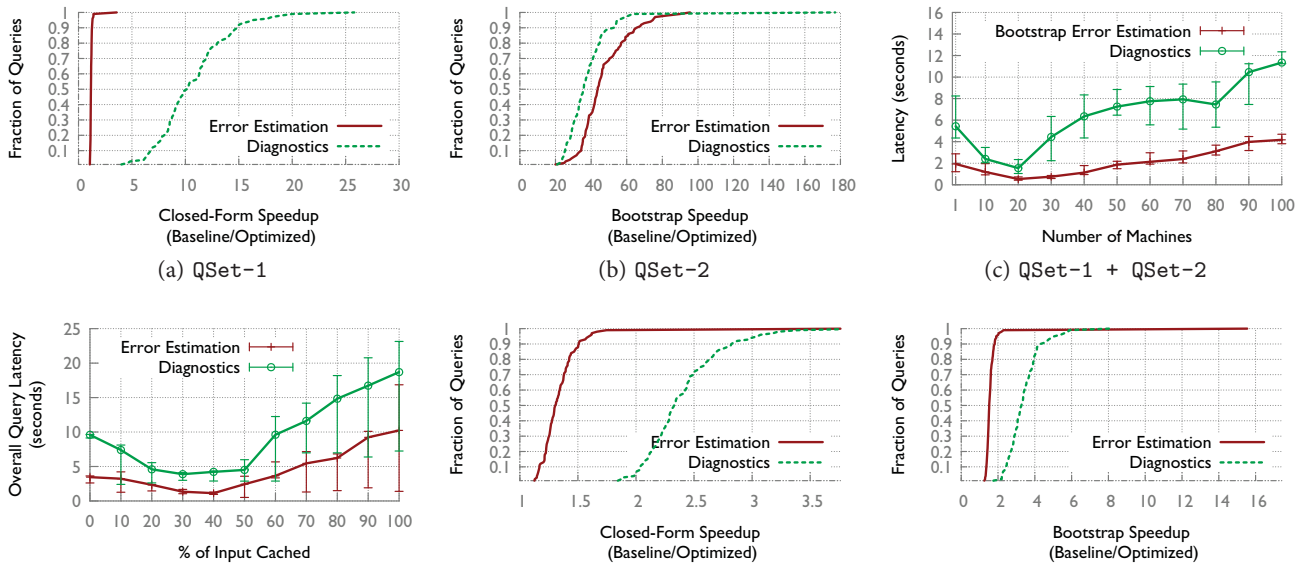
Similarly, we observed that caching the entire set of input samples is not always optimal. As we explain in §6, given that the total RAM in a cluster is limited, there is a tradeoff between caching a fraction of input data versus intermediate data during query execution. In



(a) QSet-1

(b) QSet-2

Figure 7: 7(a) and 7(b) show the naive end-to-end response times and the individual overheads associated with query execution, error estimation and diagnostics for QSet-1 (*i.e.,* set of 100 queries which can be approximated using closed-forms) and QSet-2 (*i.e.,* set of 100 queries that can only be approximated using bootstrap) respectively. Each set of bars represents a single query execution with a 10% error bound.



(a) QSet-1 (b) QSet-2 (c) QSet-1 + QSet-2

(d) QSet-1 + QSet-2 (e) QSet-1 (f) QSet-2

Figure 8: Fig. 8(a) and Fig. 8(b) show the cumulative distribution function of speedups yielded by query plan optimizations (*i.e., Scan Consolidation* and *Sampling Operator Pushdown*) for error estimation and diagnostics with respect to the baseline defined in §5.2. Similarly, Fig. 8(e) and Fig. 8(f) show the speedups yielded by a fine grained control over the physical plan (*i.e.,* bounding the query's *degree of parallelism*, *size of input caches* and *mitigating stragglers*) for error estimation and diagnostics with respect to the baseline defined in §5.3. Fig. 8(c) and Fig. 8(d) demonstrate the trade-offs between the bootstrap-based error estimation/diagnostic techniques and the number of machines or size of the input cache respectively (averaged over all the queries in QSet-1 and QSet-2 with vertical bars on each point denoting 0.01 and 0.99 quantiles).

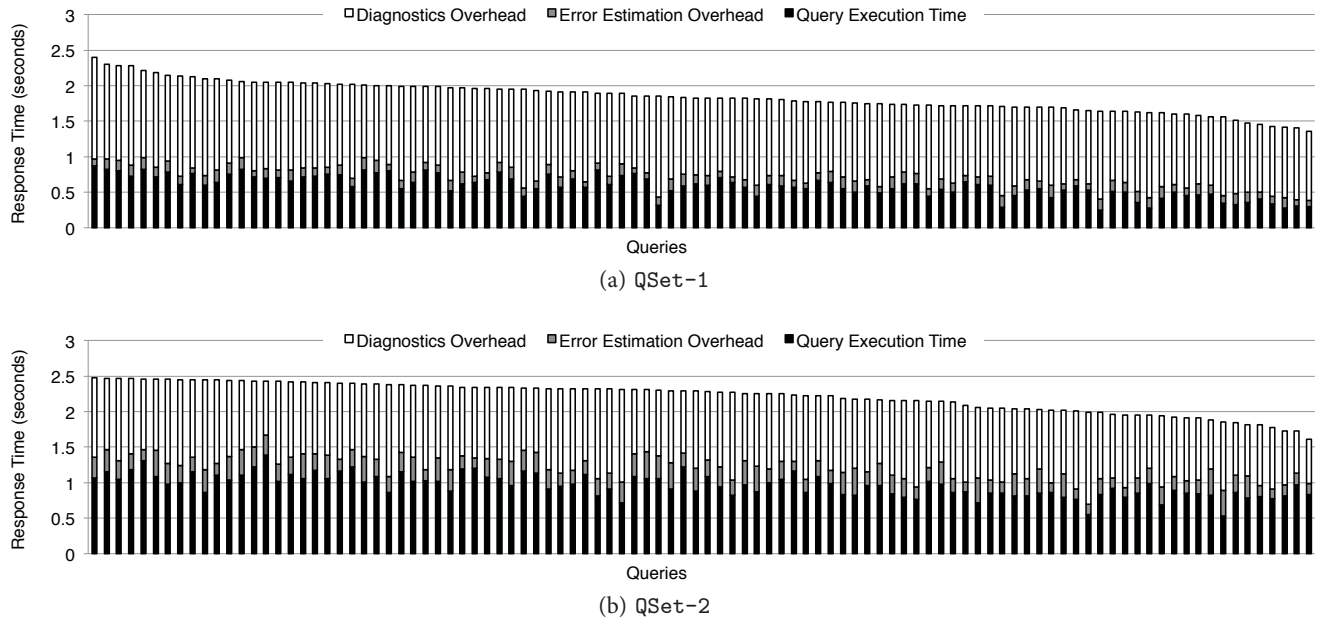


Figure 9: Fig. 9(a) and Fig. 9(b) show the optimized end-to-end response times and the individual overheads associated with query execution, error estimation and diagnostics for `QSet-1` (*i.e.,* set of 100 queries which can be approximated using closed-forms) and `QSet-2` (*i.e.,* set of 100 queries that can only be approximated using bootstrap) respectively. Each set of bars represents a single query execution with a 10% error bound.

our experiments we observed that caching a fixed fraction of samples and allotting a bigger portion of memory for query execution results in the best average query response times. Fig. 8(d) shows the tradeoff graph with the percentage of samples cached on the x-axis and the query latency's on the y-axis. In our case, we achieve the best end-to-end response times when 30 – 40% of the total inputs were cached, accounting for roughly 180 – 240 GB of aggregate RAM.

Fig. 8(e) and Fig. 8(f) show the cumulative distribution function of the speedup for error estimation and diagnostics obtained by (i) tuning the degree of parallelism, (ii) the fraction of samples being cached (as discussed above), and, in addition, (iii) increasing the number of sub-queries by 10% to account for straggler mitigation. Specifically, 8(e) and 8(f) show the speedups associated with `QSet-1` (*i.e.,* set of 100 queries that can be approximated using closed-forms) and `QSet-2` (*i.e.,* set of 100 queries that can only be approximated using bootstrap) respectively. Note that the baseline for these experiments is the implementation of BlinkDB with the query plan optimizations described in §5.3, and not the naive implementation.

7.4 Putting it all Together

With all the optimizations in place, Fig. 9(a) and Fig. 9(b) show the per-query overheads of error estimation and diagnostics on our two query sets `QSet-1` and `QSet-2`, respectively. Note that in comparison to Fig. 7(a) and Fig. 7(b), we have been able to improve the execution times by 10 – 200×, and achieve end-to-end response times of a few seconds, thus effectively providing interactivity.

8. CONCLUSION

Sampling produces faster response times, but requires error estimation. Error estimation can be performed via the bootstrap or closed forms. Both techniques work often enough that sampling is worthwhile, but they fail often enough that a diagnostic is required. In this paper, we first generalized the diagnostic (that was previously applied to the bootstrap) to other error estimation techniques (such as closed forms) and demonstrated that it can identify most failure cases. For these techniques to be useful for interactive approximate query processing, we require each of the aforementioned steps to be extremely fast. Toward this end, this paper proposes a series of optimizations at multiple layers of the query processing stack. With fast error estimation and diagnosis of error estimation failure, significantly faster response times are possible, making a qualitative difference in the experience of end users.

Acknowledgements

The authors would like to thank Purnamrita Sarkar and the members of the Facebook Presto team for their invaluable feedback and suggestions throughout this project. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, LBNL Award 7076018, and DARPA XData Award FA8750-12-2-0331, gifts from Amazon Web Services, Google, SAP, The Thomas and Stacey Siebel Foundation, Apple Inc., Cisco, Cloudera, EMC, Ericsson, Facebook, GameOnTalis, Guavus, Hortonworks, Huawei, Intel, Microsoft, NetApp, Pivotal, Samsung, Splunk, Virdata, VMware, WANdisco and Yahoo!, and fellowships from Qualcomm and Facebook.

9. REFERENCES

[1] AMPLab Big Data Benchmark. `https://amplab.cs.berkeley.edu/benchmark/`.
[2] BlinkDB alpha 0.1.1. `http://blinkdb.org`.
[3] Common Crawl Document Corpus. `http://commoncrawl.org/`.
[4] Conviva Networks. `http://www.conviva.com/`.
[5] Facebook Presto. `http://prestodb.io/`.
[6] Intel Hadoop Benchmark. `https://github.com/intel-hadoop/HiBench`.
[7] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In *VLDB*, September 1999.
[8] 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 *EuroSys*, pages 29–42, 2013.
[9] S. Agarwal, A. Panda, B. Mozafari, A. P. Iyer, S. Madden, and I. Stoica. Blink and It's Done: Interactive Queries on Very Large Data. *PVLDB*, 5(12):1902–1905, 2012.

[10] P. Agrawal, D. Kifer, and C. Olston. Scheduling shared scans of large data files. *PVLDB*, 2008.

[11] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou. Recurring job optimization in scope. In *SIGMOD Conference*, pages 805–806, 2012.

[12] A. J. Canty, A. C. Davison, D. V. Hinkley, and V. Ventura. Bootstrap diagnostics and remedies. *Canadian Journal of Statistics*, 34(1):5–27, 2006.

[13] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.

[14] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.

[15] A. Dobra, C. Jermaine, F. Rusu, and F. Xu. Turbo-charging estimate convergence in dbo. *PVLDB*, 2(1):419–430, 2009.

[16] B. Efron. Bootstrap methods: another look at the jackknife. *The annals of Statistics*, pages 1–26, 1979.

[17] B. Efron and R. Tibshirani. *An introduction to the bootstrap*, volume 57. CRC press, 1993.

[18] G. Giannikis, G. Alonso, and D. Kossmann. Shareddb: Killing one thousand queries with one stone. *PVLDB*, 2012.

[19] P. J. Haas and P. J. Haas. Hoeffding inequalities for join-selectivity estimation and online aggregation. In *IBM Research Report RJ 10040, IBM Almaden Research*, 1996.

[20] P. Hall. On symmetric bootstrap confidence intervals. *Journal of the Royal Statistical Society. Series B (Methodological)*, 50(1):pp. 35–45, 1988.

[21] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

[22] A. Kleiner, A. Talwalkar, S. Agarwal, I. Stoica, and M. I. Jordan. A general bootstrap performance diagnostic. In *KDD*, pages 419–427, 2013.

[23] N. Laptev, K. Zeng, and C. Zaniolo. Early Accurate Results for Advanced Analytics on MapReduce. *PVLDB*, 5(10):1028–1039, 2012.

[24] C. McDiarmid. Concentration. In *Probabilistic methods for algorithmic discrete mathematics*, pages 195–248. Springer, 1998.

[25] R. B. Miller. Response time in man-computer conversational transactions. In *Proceedings of the December 9-11, 1968, fall joint computer conference, part I*, pages 267–277. ACM, 1968.

[26] B. Mozafari and C. Zaniolo. Optimal load shedding with aggregates and mining queries. In *ICDE*, 2010.

[27] F. Olken and D. Rotem. Simple random sampling from relational databases. In *VLDB*, pages 160–169, 1986.

[28] N. C. Oza. Online bagging and boosting. In *International Conference on Systems, Man, and Cybernetics, Special Session on Ensemble Methods for Extreme Environments*, pages 2340–2345, 2005.

[29] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4(11):1135–1145, 2011.

[30] A. Pol and C. Jermaine. Relational confidence bounds are easy with the bootstrap. In *In SIGMOD Conference Proceedings*, 2005.

[31] C. Qin and F. Rusu. Pf-ola: a high-performance framework for parallel online aggregation. *Distributed and Parallel Databases*, pages 1–39, 2013.

[32] J. Rice. *Mathematical Statistics and Data Analysis*. Brooks/Cole Pub. Co., 1988.

[33] A. Thusoo et al. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.

[34] A. van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 2000.

[35] A. van der Vaart and J. Wellner. *Weak Convergence and Empirical Processes: With Applications to Statistics*. Springer Series in Statistics. Springer, 1996.

[36] X. Wang, C. Olston, A. D. Sarma, and R. C. Burns. Coscan: cooperative scan sharing in the cloud. In *SoCC*, 2011.

[37] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: SQL and rich analytics at scale. In *SIGMOD Conference*, pages 13–24, 2013.

[38] M. Zaharia et al. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*, 2012.

[39] K. Zeng et al. The Analytical Bootstrap: a New Method for Fast Error Estimation in Approximate Query Processing. In *SIGMOD*, 2014.

APPENDIX

A. DIAGNOSTIC ALGORITHM

Algorithm 1 is the diagnostic of Kleiner et al [22], specialized to query approximation. We provide it here for completeness. The algorithm requires a large number of parameters. In our experiments, we have used settings similar to those suggested by Kleiner et al.: $p = 100$, $k = 3$, $c_1 = 0.2$, $c_2 = 0.2$, $c_3 = 0.5$, and $\beta = 0.95$ on subsamples whose rows have total size $50MB$, $100MB$ and $200MB$.

Input: $S = (S_1, ..., S_n)$: a sample of size n from D
θ: the query function
α: the desired coverage level for confidence intervals
ξ: a function that produces confidence interval estimates given a sample, a query function, and a coverage level, e.g. the bootstrap
$b_1, ..., b_k$: an increasing sequence of k subsample sizes
p: the number of simulated subsamples from D at each sample size
c_1, c_2, c_3: three different notions of acceptable levels of relative deviation of estimated error from true error
ρ: the minimum proportion of subsamples on which we require error estimation to be accurate
Output: a boolean indicating whether confidence interval estimation works well for this query

`// Compute the best estimate we can find for` $\theta(D)$`:`
$t \leftarrow \theta(S)$
for $i \leftarrow 1...k$ **do**
  $(\mathbb{S}^{i1}, \mathbb{S}^{i2}, ..., \mathbb{S}^{ip}) \leftarrow$ any partition of S into size-b_i subsamples
  `// Compute the true confidence interval for subsample size` b_i`:`
  $(\hat{t}_{i1}, ..., \hat{t}_{ip}) \leftarrow \text{map}(\mathbb{S}^{i\bullet}, \theta)$
  $x_i \leftarrow$the smallest symmetric interval around $\theta(S)$ that covers αp elements of $\hat{t}_{i\bullet}$
  `// Compute the error estimate for each subsample of size` b_i`. (Note that when` ξ `is the bootstrap, this step involves computing` θ `on many bootstrap resamples of each` $\mathbb{S}^{ij}$`.)`
  $(\hat{x}_{i1}, ..., \hat{x}_{ip}) \leftarrow \text{map}(\mathbb{S}^{i\bullet}, s \mapsto \xi(s, \theta, \alpha))$
  `// Summarize the accuracy of the error estimates` $\hat{x}_{i\bullet}$ `with a few statistics: Magnitude of average deviation from` x_i `(normalized by` x_i`), spread (normalized by` x_i`), and proportion acceptably close to` x_i`:`
  $\Delta_i \leftarrow \left|\frac{\text{mean}(\hat{x}_{i\bullet}) - x_i}{x_i}\right|$
  $\sigma_i \leftarrow \frac{\text{stddev}(\hat{x}_{i\bullet})}{x_i}$
  $\pi_i \leftarrow \frac{\text{count}(j : |\frac{\hat{x}_{ij} - x_i}{x_i}| \leq c_3)}{p}$
end
`// Check several acceptance criteria for the` $\hat{x}_{ij}$`. Average deviations and spread must be decreasing or small, and for the largest sample size` b_k `most of the` $\hat{x}_{k\bullet}$ `must be close to` x_k`:`
for $i \leftarrow 2...k$ **do**
  $\text{AverageDeviationAcceptable}_i \leftarrow (\Delta_i < \Delta_{i-1} \text{ OR } \Delta_i < c_1)$
  $\text{SpreadAcceptable}_i \leftarrow (\sigma_i < \sigma_{i-1} \text{ OR } \sigma_i < c_2)$
end
$\text{FinalProportionAcceptable} \leftarrow (\pi_k \geq \rho)$
return `true` if $\text{AverageDeviationAcceptable}_i$ AND $\text{SpreadAcceptable}_i$ for all i AND FinalProportionAcceptable, and `false` otherwise

Algorithm 1: The diagnostic algorithm of Kleiner et al. We use functional notation (*e.g.,* "map") to emphasize opportunities for parallelism.