Bulletin of the Technical Committee on

Data Engineering

September 2015 Vol. 38 No. 3

IEEE Computer Society

Letters

Special Issue on Approximate Query Processing and Applications

Conference and Journal Notices

The TC on Data Engineering

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems. The TCDE web page is http://tab.computer.org/tcde/index.html.

The Data Engineering Bulletin

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

The Data Engineering Bulletin web site is at http://tab.computer.org/tcde/bull_about.html.

Letter from the Editor-in-Chief

Evolution of the Bulletin

Over the years, the Bulletin has remained remarkably stable in its mission of informing readers about the state of ongoing work in important research areas– and making sure that best industry practices are made more widely known to the database research community. What has not remained completely stable is the infrastructure and what it enables in terms of delivering the contents of the Bulletin to readers.

The most recent example of this is that the program dvipdfm is now used to generate not only the full issue, but also individual papers of the issue. This resulted in my experiencing a "learning curve" (helped by Sudarshan) and required some modification of existing latex style files and tex files to accommodate this. The result is that now individual papers have correct page numbering. And this has been applied retro-actively back through 2010. My guess is that mostly authors will be sensitive to this, but that readers will notice it.

My expectation is that this kind of infrastructure evolution will continue, with resulting benifits for both authors and readers. If you have any suggestions about this, please let me know. The Bulletin is a continous work-in-progress. Having reader input in this would be extremely useful.

The Current Issue

We now live in what might be called a "post-SQL" world. SQL continues to be used widely. But many other avenues are being explored. This has been the result of living in a "post-DBMS" world. Again, not because DBMSs are no longer important. Quite the contrary. But what is new is that the domain of interest to query processing has expanded enormously- as the size of data subject to query processing has grown explosively.

This new world of dealing also with "big data" has seen the emergence of analytic engines, some based on map-reduce, some on streaming systems, etc. These systems typically are scan-oriented, processing enormous volumes of data at high speed. But if the data is large enough, even "high speed" may not produce results with an appropriately useful latency. Even were the software perfect, memory bandwidth provides serious limitations on system responsiveness.

The quest for useful data is the driver for many ML efforts. There is the intuition that if we can just find the data (assumed to be "out there", then we can solve interesting problems, both for industry and government.

Both these scenarios require a fresh look at data management. Enter "approximate query processing". This is not your father's Oldsmobile, or even your father's query processor. Though work has been going on that has nibbled at this space, this is an area that has bloomed in the current era. Thus, this is an excellent time to take a look at how work is proceeding in approximate query processing. This is, of course, a snapshot of the state-of-the-art. Work is ongoing in this area, with active, insightful, and important work continuing. Chris Jermaine has put together a very useful snapshot of the kinds of work that is being done in the research community. The issue is both an introduction and a survey of this area, with technology that is well worth knowing about, given the importance of the area. I want to thank Chris for serving as editor for the issue. It is well worth reading. This area will only grow in importance.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Approximate query processing (AQP) has a long history in databases—for at least 30 years, people have been interested in trading accuracy for better performance, with "better performance" equating to faster response time or lower memory utilization. Over the years, various approximation methodologies have been considered, including sampling, histograms, wavelets, and sketches.

Curiously, those decades of high-quality academic and industrial research on approximation have had surprisingly little commercial impact, at least until recently. While databases are often sampled, direct support for AQP in data management systems is not widespread. There are plenty of explanations for this: one can argue that few people had sufficient data that performance was enough of a concern to accept approximation; end users were suspicious of approximations and statistical methods; database approximation lacked a "killer app" where approximation was absolutely necessary in order to obtain an answer.

There is reason to believe, however, that this lack of adoption of AQP techniques may soon end. The advent of the "Big Data" era and the importance of analytics in general means that very large data sets are ubiquitous. Further, cloud computing means that data processing is now often pay-as-you-go, so less accuracy may mean less cost. Statistics and data science are everywhere. Data mining and machine learning are inherently statistical endeavors—approximation is often vital to scaling such algorithms. If applications such as crowdsourcing become important, approximation is mandatory.

In this issue, we have a slate of very interesting articles on modern applications of AQP as well as modern AQP methodologies. In the issue's introductory paper, "A Handbook for Building an Approximate Query Engine," Mozafari and Niu describe many of the fundamentals that must be understood to build a modern approximate query processing engine. In the second paper, "Scalable Analytics Model Calibration with Online Aggregation," Florin Rusu and his co-authors describe how a now-classical approach to AQP called "online aggregation" can be used to help solve an important problem in large-scale statistical learning: controlling the speed of convergence of gradient descent algorithms. In "On the Complexity of Evaluating Order Queries with the Crowd," Groz, Milo, and Roy consider a case where AQP is mandatory: when crowdsourcing must be used to discover the answer to questions that cannot be answered by simply scanning a database.

In the paper entitled "SampleClean: Fast and Reliable Analytics on Dirty Data," Krishnan and his co-authors describe how AQP can be used to perform in-database analytics when some of the data stored are inaccurate: rather than cleaning all of the data, a small minority can be cleaned and the result of cleaning the rest approximated. In "Independent Range Sampling on a RAM," Hu, Qiao, and Tao investigate the problem of how to actually draw the samples that are needed to power the approximation provided by a sampling-based query: how can we efficiently draw a uniform random sample from a database range query?

In "Hidden Database Research and Analytics (HYDRA) System," Lu and co-authors describe their research in using sampling to answer queries over "hidden databases," or those that are accessible only through a limited user interface (typically on the web), and cannot be examined directly. Finally, in "Approximate Geometric Query Tracking over Distributed Streams," Minos Garofalakis describes how complex queries can be tracked over distributed data streams using the Geometric Method. In such a situation, AQP may be mandatory as it is impossible to move all data to a single location for exact processing.

I hope that you enjoy the issue as much as I enjoyed putting it together!

Chris Jermaine
Rice University

A Handbook for Building an Approximate Query Engine

Barzan Mozafari Ning Niu
University of Michigan, Ann Arbor
{mozafari,nniu}@umich.edu

Abstract

There has been much research on various aspects of Approximate Query Processing (AQP), such as different sampling strategies, error estimation mechanisms, and various types of data synopses. However, many subtle challenges arise when building an actual AQP engine that can be deployed and used by real world applications. These subtleties are often ignored (or at least not elaborated) by the theoretical literature and academic prototypes alike. For the first time to the best of our knowledge, in this article, we focus on these subtle challenges that one must address when designing an AQP system. Our intention for this article is to serve as a handbook listing critical design choices that database practitioners must be aware of when building or using an AQP system, not to prescribe a specific solution to each challenge.

1 Introduction

We start this article with a discussion of when and why AQP is a viable solution in Section 2. After a brief overview of the typical anatomy of AQP systems in Section 3, we discuss the common sampling techniques used in practice in Section 4. Often, database developers are not aware of the subtle differences between different sampling strategies and their implications on the error estimation procedures. While confidence intervals are commonly used for quantifying the approximation error, in Section 5 we discuss other important types of error (i.e., subset and superset errors) that are not expressible as confidence intervals, but are quite common in practice. We then discuss another source of error, *bias*, in Section 6. Surprisingly, while well-studied and understood by statisticians, bias seems to be often overlooked in the database literature on AQP systems. The complexity and variety of different types of errors can easily render an AQP system incomprehensible and unusable for an average user of database systems. Intuitive but truthful communication of error with the end user is discussed in Section 7. We conclude this article in Section 8.

2 Why use Approximation in the First Place?

Since its inception 25 years ago [63], Approximate Query Processing (AQP) has always been an active area of research in academia [7, 11, 20, 9, 32, 53, 58]. However, AQP has recently gained substantial attention in the commercial world as well. Soon after the commercialization of BlinkDB by Databricks [1], other Big Data query engines also started to add support for approximation features in their query engines (e.g., SnappyData [5]

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

or Facebook's Presto [4]). In the rest of this section, we try to explain the reason, rationale, and justification behind this commercial interest.

2.1 The Losing Battle for Interactive Response Times

Online services, wireless devices, and scientific simulations are all creating massive volumes of data at unprecedented rates. This abundance of rich datasets has made *data-driven decisions* the predominant approach across businesses, science, and even governments. Ironically, existing data processing tools have now become the bottleneck in data-driven activities. When faced with sufficiently large datasets (e.g., a few terabytes), even the fastest database systems can take hours or days to answer the simplest queries [11]. This response time is simply unacceptable to many users and applications. The exploratory data analytics is often an *interactive* and *iterative* process: a data scientist forms an initial hypothesis (e.g., by visualizing the data), consults the data, adjusts his/her hypothesis accordingly, and repeats this process until a satisfactory answer is discovered. Thus, slow and costly interactions with data can severely inhibit a data scientist's productivity, engagement, and even creativity.[1]

Driven by the growing market for interactive analytics, commercial and open source data warehouses (e.g., Teradata, HP Vertica, Hive, Impala, Presto, Spark SQL, Redshift) have entered an arms race towards providing interactive response times through implementing various optimization techniques. These optimization range from exploiting parallelism, to indexing, materialization, better query plans, data compression, columnar formats, and in-memory and in-situ processing. In essence, these optimizations are no different than the mainstream database research on query processing over the past four decades, where the goal has been simply to:

1. efficiently access *all* relevant tuples to the query while avoiding the irrelevant tuples (e.g., via indexing, materialization, compression, caching); and

2. choose a query plan with the least cost among a set of alternative plans that are all *equivalent* (i.e., all plans are correct).

As long as we pursue these utopian goals,[2] all our query optimization techniques are eventually destined to fail. This is because the growth rate of data has already surpassed Moore's law [3]. It is true that the set of tuples queried by the user, called the *working set*, is often significantly smaller than the entire data, e.g., queries might focus on the last week worth of data. However, even the data generated and collected each week is growing at the same speed. This means that even if we know which exact tuples are accessed by a query in advance, at some point, the number of those tuples will be large enough that they will no longer fit in memory (even when compressed). In other words, since memory and CPU improvements are lagging *exponentially* behind the rate of data growth, our ability to provide interactive response times will only decrease. Eventually, when the datasets become large enough, no optimization technique will be able to provide interactive response times unless we relax our goal of processing *all* relevant tuples.

Therefore, AQP—whereby only a small fraction of the relevant tuples are processed—presents itself as the only viable approach to ensure interactive response times in the near future.

2.2 Perfect Decisions are Possible even with Imperfect Answers

In many applications, query results are only useful inasmuch as they enable best decisions, as exemplified by the following use cases.

[1] For instance, studies have shown that to maintain a human's interactive engagement with a computer, the response time must not exceed 2–10 seconds [44].

[2] Our discussion here is focused on relaxing the first goal. For a discussion on relaxing the second goal refer to [46].

A/B testing —. The goal in A/B testing is to compare two quantities $f(A)$ and $f(B)$ for some assessment function f, and thereby choose the better explanation. However, as long as our estimates of these quantities reliably allow for such comparisons, there is no need for computing their exact values. For instance, as long as our approximations guarantee with high confidence (e.g., 99%) that $f(A) \in [0.1, 0.2]$ while $f(B) \in [0.5, 0.8]$, there is no need for knowing the exact values of those two quantities.

Hypothesis testing — A major goal of data scientists is to evaluate and test a set of competing hypotheses and choose the *best* one. Choosing the best hypothesis often involves comparing the effectiveness scores of different hypotheses. The effectiveness score of a hypothesis can be measured by computing some function of the hypothesis, e.g., probability values (p-values), the generalization-error, confidence intervals, occurrence ratios, or reduction of entropy. These scores are initially only useful insofar as they allow for ruling out the unlikely hypotheses in light of the more likely ones. In other words, scientists do not need the precise value of these scores. As long as we can guarantee that a hypothesis yields a lower score than others, the researchers can proceed to choose the better hypothesis.

Root cause analysis — Scientists and data analysts frequently seek explanations for or identify the root causes of various observations, symptoms, or faults. Similar to previous cases, root cause analysis can be accomplished without having precise values.

Exploratory analytics — Data scientists often slice and dice their dataset in search of interesting trends, correlations or outliers. For these exploration purposes, receiving approximate answers at interactive speeds is preferred over exact and slow results. In fact, research on human-computer interaction has shown that, to keep users engaged, the response time of an interactive system must be below 10 seconds. In particular, during creative efforts, the idle time beyond a couple of seconds, while the user waits to see the consequence of his/her query, is inhibitive and intolerable [44].

Feature selection — An important step in training high-quality models in machine learning consists of searching the space of possible subsets of features. The exponential number of possibilities has made feature selection one of the most expensive (yet crucial) steps in a machine learning workflow. Similar to the previous examples, as long as a set of features can be assessed as inferior to another set, the former can be safely ruled out without having to compute the exact prediction error of the model under these subsets of features (e.g., see [38]).

Big data visualization —Interactive visualization is a key means through which analysts discover interesting trends in scientific or business-related data. Modern visualization tools, such as Tableau [6] and GoodData [2], use external database engines to manage large datasets. When the user requests a plot, the visualization tool submits a query to the database engine to fetch the data needed for the visualization. Unfortunately, for large datasets it may take the system several minutes or even hours to obtain the tuples necessary for generating a plot. Increasingly, the explicit goal of visualization systems is not just to generate a single precise image, but to enable a fluid back-and-forth engagement between the user and the dataset [41]. When the database system fails to respond in a short amount of time, this fluid interactivity is lost. However, loading (and plotting) more data does not always improve the quality of the visualization. This is due to three reasons [55]: (i) the finite number of pixels on the screen, (ii) the cognitive limitations of human vision in noticing small details, and (iii) the redundancy in most real-world datasets. Thus, it is often possible for the database system to generate a degraded query answer using a small sample of the large dataset but with minimal impact on the visualization's quality [27, 37, 55].

2.3 The Data Itself is Often Noisy

Most of the data collected in the real world is inherently noisy, due to human error, missing values, white noise, sensor misreadings, erroneous data extractors, data conversions and even rounding errors. In other words, *precise* answers are often a myth: even when processing the entire data, the answer is still an approximate one.

Due to these reasons, a significant body of research in learning theory has accounted for this inherent noise when performing inference using training data. In fact, there has been much research on trade-off between inference error and computation [18]. Likewise, introducing a controlled degree of noise by database systems in exchange for significant savings in computational resources is quite natural.

2.4 Inference and Predictive Analytics are Probabilistic in Nature

The majority of the commonly used techniques for inference in machine learning, computational statistics, and stochastic modeling are based on probabilistic analysis. In some cases, the predictions themselves are probabilistic (e.g., in probabilistic regression [17]), while in other cases, the predictions are based on probabilistic reasoning. As these predictive models are designed to account for uncertainty and noise in their training data, adding a controlled degree of uncertainty to their input (e.g., by using a carefully selected sample) will not necessarily break their logic. In many cases, using a smaller sample of the data will lead to significant computational savings with no or little impact on the utility and accuracy of their predictions. This is because the time complexity of many learning algorithms is super-linear (e.g., cubic or quadratic) in their input size, while the improvement of their learning curve follows a diminishing return rule. In other words, the learning error is often proportional to $O(\frac{1}{\sqrt{n}})$ where n is the number of tuples used for training, e.g., to reduce the error from 20% to 2%, one typically needs to use $100\times$ more data (see Corollary 3.4 in [45]). In fact, in some cases, certain forms of data reduction can significantly improve accuracy, e.g., by sampling down the majority class and creating a more balanced distribution of different labels for classification tasks [40]. In general, sampling is a systematic way of reducing the data size while maintaining the essential properties of the data [43]. For instance, with proper sampling, the solution to the sub-sampled problem of linear regression is also a good approximate solution to the original problem [24].

Therefore, in many cases where the output of database queries are consumed by predictive analytics or machine learning algorithms, returning samples of the data or approximate results is a sound decision that can yield considerable performance benefits.

3 General Anatomy of Approximate Query Processing Systems

AQP systems can widely differ in their methodology and design choices. However, in this section, we attempt to provide a general anatomy by identifying the common architectural components of these systems. Our discussion of the various design decisions in each component will also serve as a natural taxonomy of the AQP systems. These components, shown in Figure 1, are discussed in the following subsections.

3.1 User Interface

This component is typically in charge of receiving queries and accuracy or latency requirements from the user, as well as communicating the approximate answers and the error bounds (or accuracy guarantees) back to the user. Some of these functionalities vary among different AQP systems. For example, systems providing an online aggregation [32, 65] interface do not take accuracy requirements from the user. Instead, they periodically report the current approximation and its estimated error as they are progressively processing the user query over a larger portion of the data. In these systems, the user can decide, based on the current accuracy of the running approximation (or the rate of its improvement), whether to terminate the query execution or await its completion over the entire data. As another exception, the user interface in certain AQP systems may not even return any error estimation to the user. These systems are often designed for debugging or real-time exploration purposes (e.g., in massive distributed systems [59] or scientific datasets [52]), and hence, the focus is on retrieving the outliers or corner cases. Here, an approximate answer—even without an accuracy guarantee—is often sufficient for a programmer or scientist to find a bug or suspect a trend in the underlying data. Once

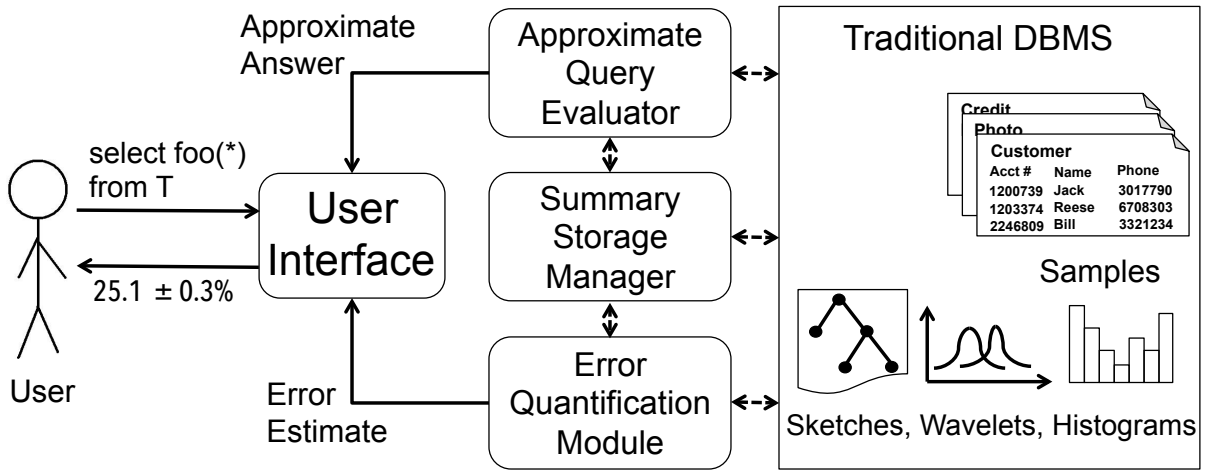


Figure 1: The general anatomy of an approximate query processing system.

a reasonable hypothesis is formed, an exact answer can be obtained by running on the entire data. However, most AQP systems allow the users to specify their accuracy or latency requirements (either on a per-query basis [12, 66] or as a system-wide setting), and report the accuracy of their results back to the user. The ability to specify and receive accuracy guarantees seems to an important factor in convincing users to use AQP systems. Min-max ranges [57], confidence intervals (a.k.a. error bars) [7, 16, 32, 66], additive errors [22], and probability of existence [23] have been used for communicating the error to the end users. However, given that the majority of database users are not statisticians, devising user-friendly and high-level mechanisms for expressing the error seems to be an area of research that requires more attention (see Section 7).

3.2 Summary Storage Manager

The summary storage manager is one of the key architectural components of an AQP system, responsible for (i) *the type of data summary* to use, (ii) *when to build* it, (iii) *how to store it*, and (iv) *when to update* it. The behavior, generality and efficiency of an AQP system depends heavily on the design choices made for how to handle these responsibilities in the summary storage manager.

Different types of data summaries — Various forms of data summaries and synopses have been utilized by AQP systems. *Samples* have been the most commonly used form of data summary. Several reasons have contributed to this popularity. By adhering to the same schema as the original table, samples support the widest range of queries compared to other types of data summaries. Besides their generality, samples are also extremely easy to implement. Most queries can be evaluated on a sample with little or no modifications to existing database engines. As one of the oldest concepts in statistics, there is also a large body of rich literature on the design and analysis of sampling-based approximations. Sampling also avoids the so called "curse of dimensionality"; the accuracy of a sample does not degrade with the the number of data dimensions. In particular, with random (a.k.a. uniform) sampling, the space required to store the sample only grows linearly with the number of attributes. Finally, most sampling algorithms do not use the AVI (Attribute Value Independence) assumption, and therefore, allow for high quality selectivity estimation [15]. Although virtually any query can be evaluated on a sample (rather than the entire dataset), the quality of the resulting approximation varies widely depending on the query. For example, evaluating `MIN`, `MAX`, top-K, and `COUNT DISTINCT` queries on a sample is often futile. Moreover, random samples are by definition inapt at capturing outliers in skewed distributions. As a

result, many AQP systems have utilized various forms of *biased* samples in addition to random samples. In particular, *stratified sampling* has been a popular choice [8, 11, 13, 20, 35]. In survey design, stratified sampling has been traditionally used for reducing the overall variance by grouping similar items into the same stratum (and hence, reducing the intra-stratum variance). However, in AQP systems, stratified sampling has been used mainly for a different reason. Queries with highly selective `WHERE` clauses may match zero or only a few tuples in a random sample, leading to extremely large (or infinite) errors. By stratifying tuples based on the value of a subset of the attributes, any combination of those columns—no matter how rare—will be preserved in the sample [8, 11, 13, 20, 35]. Thus, a key challenge in using stratified samples is to decide which subsets of columns to stratify on. In other words, finding an optimal set of stratified samples is a physical design problem: given a fixed storage budget, choose subsets of columns that are most skewed, or are most likely to be filtered on by future queries (see [11] for a mathematical formulation of this problem). The choice of optimal set of stratified and random samples depends heavily on the specific queries that users want to run. However, since future queries are not always known *a priori* (especially, for exploratory and adhoc workloads [48]), special care needs to be taken to ensure that the selected samples remain optimal even when future workloads deviate from past ones (see [47] for a solution).

Histograms are another form of data summaries commonly used by AQP systems. A histogram is a binned representation of the (empirical) distribution of the data. One-dimensional (i.e., single-column) histograms are internally used by nearly all database systems to cost and selectivity estimation of different query plans. Since AQP systems are often built atop an existing DBMS, these histograms can also be used for producing fast approximate answers without incurring an additional storage overhead. The main drawback of histograms is that they suffer from the curse of dimensionality; the number of bins for k-column histograms grows exponentially with k. Using the AVI assumption to estimate the joint distribution of k columns from the 1-column histograms is equally futile (due to intra-column correlations).

Wavelet transforms are another means for summarizing large datasets, whereby a small set of wavelet coefficients can serve as a concise synopsis of the original data. While wavelets can exploit the structure of the data to provide considerable data compaction, they suffer from the same limitation as histograms, i.e., curse of dimensionality. In other words, one-dimensional wavelet synopses can only support a limited set of database queries. Moreover, the error guarantees traditionally offered by wavelet synopses are not always meaningful in an AQP context (see [30]).

There are numerous other types of synopses structures that can be built and used by AQP systems, including but not limited to sketches, coresets, and ϵ-nets. A detailed overview of various types of synopses is beyond the scope of this article.[3] While some of these data summaries are more practical than others, their common theme is that they are all specific to a certain type of query, but provide significantly better approximations than sampling-based techniques. Therefore, these types of synopses are typically used when users demand a specific query to be executed quickly and when there is an existing technique to support that type of query. Examples include HyperLogLog for `COUNT DISTINCT` queries [28], Count-Min-Sketch (CMS) for `COUNT-GROUP BY` queries [22, 42], Join Synopsis for foreign-key join queries [9], balanced box-decomposition (BBD) trees for range queries [14], and Neighbor Sensitive Hashing (KSH) tables for k-nearest neighbor queries [54].

The choice of the specific data summary depends on the target workload. AQP systems designed for speeding up a predetermined set of queries utilize various sketches, wavelets or histograms, whereas general-purpose AQP systems tend to rely on sampling strategies. Often, a combination of these data summaries are built to provide general support for most queries, while producing better approximations for those queries that are more important to the user. As a rule of thumb, building a random sample, several stratified samples, as well as a few common sketches is a reasonable choice.

Offline versus online data summarization — Another choice made by the Summary Storage Manager is whether to summarize the data prior to the arrival of the query (*offline*) or perform the summarization on demand

[3] Interested readers are referred to [21] for an excellent and detailed discussion of different types of synopses.

after receiving the query (*online*). Performing the data summarization offline allows the AQP system to utilize expensive, more sophisticated types of synopses, e.g., summaries that require multiple passes over the data. At the same time, the generated summaries can be further optimized through various physical layout strategies, such as replication, indexing, partitioning and caching (see below). However, there is a potential downside to the offline approach: the lack of precise knowledge of future queries can lead the Summary Storage Manager to making sub-optimal choices, e.g., by not building an appropriate stratified sample. In contrast, the online approach does not have to make any choices *a priori*. However, efficient data summarization in an online fashion is significantly more challenging. In other words, the online approach will incur an additional run-time overhead and will have fewer opportunities for building sophisticated types of synopses or auxiliary structures on demand.

Different storage strategies — The Summary Storage Manager is also in charge of the physical layout of the generated summaries. For example, this component may choose to store certain summaries at multiple granularities. For instance, it may physically store three samples of size $1\%, 2\%$ and 4%, respectively, to provide three logical samples of size $1\%, 3\%$ and 7%. The advantage of such logical samples, called *overlapping samples*, is that once the query is run on a 1% sample, its results on a 3% sample can be obtained by simply performing a delta computation using the 2% sample. This strategy is particularly useful when the initially chosen sample size does not meet the accuracy requirements of the user. An alternative strategy is to use each sample individually. Given the same storage budget, this choice provides a finer range of sample size selections, e.g., if the 1% size is deemed insufficient, the next choice will be 2% rather than 3%. However, delta computation will no longer be an option in this case. For most queries, delta computation is typically faster than a re-evaluation on the entire data. However, for non-monotonic or nested queries with `HAVING` clauses, this may not be the case [65].

Most AQP systems use an existing RDBMS to store their summaries, e.g., samples of a table are themselves stored as tables. However, the Summary Storage Manager can still decide which samples need to be cached in memory and which ones should be kept on disk [11]. For example, for time series data, the most recent window might be more likely to be queried, and hence, should be kept in memory. When on disk, the columnar or row-wise layout of a sample will affect its I/O cost as well as its ease-of-update. Whether and how to index each sample is yet another choice. In distributed AQP systems, partitioning and replication strategies applied to each sample will also have significant performance implications. For example, replicating a popular sample on every node may be a reasonable strategy unless it needs to be updated frequently.

Different maintenance strategies — There are several alternatives for the maintenance of the different synopses and summaries in an AQP system. In a *lazy* approach, the summaries are only updated as a batch, either periodically (e.g., once a week) or upon the user's explicit request. This approach is often preferred when the cost of updates is prohibitive, while the downside is the obvious staleness of the approximate results until the next update is performed. In an *eager* approach, the updates to the database are immediately reflected in the data summaries. For example, whenever a tuple is inserted to or deleted from a table, all samples on that table are also updated accordingly. Typically, data summaries are easier to update in face of insertions than deletions. As a result, most AQP systems only allow insert operations, which is sufficient for append-only environments, such as data streams or HDFS-based systems (e.g., Hive, Presto, Impala, SparkSQL).

3.3 Approximate Query Evaluator

This component is perhaps the heart of every AQP system, computing approximate answers for queries received through the User Interface. The overwhelming majority of the existing AQP systems are built atop an existing database system, for two main reasons: (i) to minimize the implementation effort as the majority of the logic from existing query processors can be rescued for approximation, and (ii) to provide users an option to always go back to the entire data and request a precise answer, if they decide to. The latter has been an important factor in improving the uptake of the AQP systems in the database market. In fact, one of the reasons behind the

popularity of sampling-based approximations (compared to other forms of data summaries) is that they can be supported with minimal changes to the existing database systems.

The Approximate Query Evaluator has to decide whether the given query is of a form that can be answered with any of the available data summaries. Adhoc synopses (e.g., sketches) are often preferred over more generic types of synopses. Otherwise, the system has to decide if and which of the existing samples can be used to best meet the user's requirement. When applicable to the query, stratified samples are preferred over random samples. Once a sample type is decided, an appropriate fraction of it is chosen based on the latency and error requirements.

Once an appropriate data summary is selected, the Approximate Query Evaluator either directly evaluates the query using that structure, or, in most cases, issues certain queries to the underlying database to evaluate the query on its behalf. When possible, the latter approach allows the AQP system to take full advantage of the mature query optimizers and the efficient implementation of off-the-shelf database systems. Certain post-processing steps, such as scaling and bias correction, may also be applied to the results before returning an approximate answer to the user.

To verify whether the user's accuracy requirements are met, the Approximate Query Evaluator needs to communicate with the Error Quantification Module, either during the query evaluation or at the end of the process. While a more sophisticated Approximate Query Evaluator can perform multi-query optimization, resource management, or load balancing, in practice most AQP systems simply rely on an underlying off-the-shelf query engine to perform these tasks for them.

3.4 Error Quantification Module

An approximate answer is as useful as its accuracy guarantees. The Error Quantification Module (EQM) is in charge of quantifying or estimating the extent of error in the approximate results. Some EQMs provide an error estimate only for the final answer [7, 11], while others provide a running error estimate as the query is progressing (i.e., online aggregation [32, 65]). The latter allow users to terminate the execution as soon as they are satisfied with the current estimate or otherwise decide to abandon their current query. However, there are few AQP systems that do not provide any error guarantees, and simply return a best-effort approximation (see [52, 59]).

Another important distinction between various error quantification strategies is their ability to *predict the error* without evaluating the query. For example, given a sample of the data, a closed-form error can predict the error of an `AVG` query using a few basic statistics, e.g., the sample size and the estimated variance of the data. On the other hand, more sophisticated error estimation strategies (e.g., simulation-based bootstrap) will only know the error once they perform an expensive computation, i.e., after evaluating the query on the sample. The ability to predict the approximation is an important factor, as it allows the AQP system to choose the best type and resolution among the available data summaries according to the latency or accuracy goals of the user. When this ability is missing, the EQM module has to resort to trial-and-error: the EQM will have to use heuristics to choose a data summary, evaluate the query on that summary, and then quantify the error to see if the results meet the accuracy requirements (otherwise, the query will be repeated on a different data summary). There are different forms of approximation error used in AQP systems, as described next.

Variance — Many AQP systems have used variance as a measure of quality for their sampling-based approximate answers. Other types of data summaries have rarely been analyzed for their variance. This is perhaps due to the large body of statistical literature on the variance of sampling and survey estimators. The work on variance estimation in the context of AQP systems can be categorized into two main approaches. The first approach [11, 19, 20, 32, 34, 9, 36, 53, 64] analytically derives closed-form error estimates for simple aggregate functions, such as `SUM`, `AVG` and `COUNT`. Although computationally appealing, analytic error quantification suffers from its lack of generality. For every new type of queries, a new formula must be derived. This derivation

is a manual process that is ad-hoc and often impractical for complex queries [56].

A second approach to error estimation is to use the so called *bootstrap*. By providing a routine method for estimating errors, bootstrap is a far more general method than closed forms [10, 30, 56]. The downside of bootstrap is its computational overhead, as hundreds or thousands of bootstrap trials are typically needed to obtain reliable estimates. However, several optimizations have been proposed for speeding up the bootstrap process (see Section 6.2).

Assuming asymptotic normality (and an unbiased answer), the variance is sufficient for computing a confidence interval for the approximate answer. Rather than reporting the variance, a confidence interval is often used to communicate the error to the end user.

Bias — Bias is another important measure of quality for approximate answers. We would like to minimize the bias in our approximations, or ideally, have unbiased answers. While the formal definition of bias can be found in Section 4, its intuitive meaning is quite simple: an unbiased approximation is one where the chances of an over-estimation is the same as that of an under-estimation. In other words, if the approximation procedure is repeated many times, the results will be centered around the true answer. While taken seriously in the statistics literature, the bias is often overlooked in Error Quantification Modules of AQP systems. Most AQP systems assume that the bias approaches zero as the sample size increases. We will show, in Section 6, that this assumption is not necessarily true.

Probability of existence — Bias and variance are used for assessing the approximation quality of a numerical estimate. However, the output of database queries is not a single number, but a relation. Due to the propagation of various forms of uncertainty during query processing, certain tuples may appear in the output of the approximate answer that will not appear in the output of the exact query (see Section 5). To assess the likelihood of this phenomenon, the EQM computes a different form of error, called *the probability of existence*. The probability of existence of an output tuple is the probability with which it would also appear in the output of the exact query. This form of error is well-studied in the context of probabilistic databases [23]. In an EQM, these probabilities can be easily computed using bootstrap [67].

Other forms of error — There are many other (less common) forms of error used by EQMs for various types of data summaries. For example, *coresets*—a carefully selected subset of the original tuples used for approximating computational geometry problems, such as as k-medians and k-means— provide multiplicative error guarantees of $1 \pm \epsilon$ [31].. In other words, coresets guarantee that $(1-\epsilon)\theta \leq \hat{\theta} \leq (1+\epsilon)$, where θ is the true answer, $\hat{\theta}$ is the approximate answer, and $\epsilon > 0$ is a pre-specified parameter chosen by the user (the smaller the ϵ, the more accurate the results but the larger the coreset size). For ϵ-nets (another subset selection approximation for geometric problems), the guarantees state that any range query containing at least $\epsilon > 0$ fraction of the original tuples will also contain at least one tuple of the selected subset. The quality of wavelet-based reconstructions is often expressed in terms of the L_2-norm error of the approximation. However, there are thresholding algorithms for Haar wavelets that minimize other forms of error, such as the maximum absolute or relative error of the approximate results (see [21] and the references within). Other AQP systems have also used absolute range guarantees [57] as a measure of accuracy. Typically, none of these error guarantees are easily convertible into one another. Consequently, most AQP systems do not combine multiple types of data summaries for answering the same query, i.e., the entire query is evaluated only using one type of data summary.

4 Not All Samples are Created Equal

Let D be our dataset (e.g., a large table), q be a query, and $q(D)$ be the result of evaluating q on the entire dataset D. In sampling-based AQP systems, the goal is to use a small sample of D, say S, to provide an approximate answer to q. This is typically achieved by running a (potentially) modified version of q, say q', on S, such that $q'(S) \approx q(D)$. In this case, q' is called an estimator for q. When $|S| << |D|$, $q'(S)$ can often be computed

much more efficiently than $q(D)$. An important characteristic of an estimator is its *bias* b, defined as follows:

$$b = \mathbb{E}_{|S|=n}[q'(S)] - q(D) \tag{1}$$

Since the expectation is always over all possible samples of size n, henceforth we omit the $|S| = n$ subscript from our expectation notation. Also, for ease of presentation, we assume that queries only return a single number (though these definitions can be easily extended to relations). An ideal estimator is one that is *unbiased*, namely:

$$b = \mathbb{E}_{|S|=k}[q'(S)] - q(D) = 0$$

Unfortunately, in general, deriving an unbiased estimator q' can be extremely difficult and adhoc. As a result, most AQP systems use a heuristic, which we call the *plug-in approach* and describe in Section 4.1. (However, we later show (in Section 6) that this plug-in approach does not guarantee an unbiased estimate and thus, an additional bias correction step is often needed.) After a brief overview of the most common sampling strategies (Section 4.2), we provide a list of estimators—along with their bias and variance closed forms—for simple aggregate functions under each strategy (Section 4.3). In particular, to the best of our knowledge, some of our closed-form estimates for conditional aggregates (i.e., aggregates with a `WHERE` clause) have not been previously derived in the literature. We list these new and existing closed-form estimates in concise tables to serve as an accessible reference for non-statisticians (e.g., practitioners and engineers) involved in implementing AQP systems.

4.1 Plug-in Approach

To provide reasonable approximations without having to modify every query or make significant changes to existing query evaluation engines, the following heuristic is used by almost all AQP systems:

1. Assign an appropriate weight w_i to each sampled tuple t_i according to its sampling rate. For example, in the original table (before sampling) the weight of every tuple is 1, whereas in a random sample comprised of 1% of the original tuples $w_i = 0.01$ for all t_i. Different tuples can have different weights, e.g., in stratified sampling (see Section 4.2) the tuples in each stratum have a different sampling rate, and hence, a different weight.

2. Modify the aggregate functions (e.g., `SUM`, `AVG`, `COUNT`, quantiles) to take these weights into account in their computation. For example, to calculate the `SUM` of attribute A in n tuples, use $\sum_{i=1}^{n} \frac{t_i \cdot A}{w_i}$ (instead of $\sum_{i=1}^{n} t_i \cdot A$), where $t_i \cdot A$ is the value of attribute A in tuple t_i. Similarly, the `AVG` of attribute A will be computed as $\frac{1}{n}\sum_{i=1}^{n} \frac{A_i}{w_i}$ (instead of $\frac{1}{n}\sum_{i=1}^{n} A_i$). `COUNT` and quantile aggregates can be modified similarly.[4] (This scaling is also known as the Horvitz-Thompson (HT) estimator [33].)

3. After the two modifications above, simply run the same query (i.e., $q' = q$) on sample S to produce an approximation of $q(D)$, namely:
$$q(D) \approx q(S) \tag{2}$$

We call this the plug-in approach since we simply plug in a sample instead of the original data but run the same query. While this approach might seem intuitive and natural, it does not necessarily produce unbiased answers (see Section 6). Next, we briefly enumerate the most commonly used sampling strategies in AQP systems.

[4] While similar weighting could also be used for `MIN` and `MAX`, the resulting estimators would still be biased for these extreme statistics.

4.2 Sampling Strategies

Given the popularity of sampling techniques in AQP systems, in this section we briefly review the most common sampling strategies employed in these systems. Subsequently, in Section 4.3, we present the closed form expressions of the error for each strategy. We refer the reader to [61] for a detailed comparison of different sampling strategies, [62] for different sampling algorithms, and [29] for a discussion of updating samples.

Fixed-size sampling with replacement — Given a fixed sample size n, we choose n tuples from D, independently and at random, and include them in our sample S. Each item can be sampled multiple times. Throughout this article, we use sets and multisets interchangeably, i.e., we allow duplicate values in a set. Thus, even though there might be duplicate tuples in S, we still have $|S| = n$. This form of sampling allows for the use of bootstrap, which is a powerful error estimation technique (see Section 6.2). However, it is rarely used in AQP systems since, given the same sample size, sampling without replacement can capture more information about the underlying data.

Fixed-size sampling without replacement — Given a fixed sample size n, we choose n tuples from D at random, and include them in our sample S. Each tuple can be sampled at most once. When $n << |D|$, sampling with replacement behaves similarly to sampling without replacement. As a result, even thought bootstrap is only designed for the former, in practice it is also used for the latter when this condition holds. When maintained for a streaming dataset (where items can be added or removed), fixed-size sampling without replacement is referred to as *reservoir sampling* [62]. Probabilistic variations of this form of sampling are also available, whereby tolerating a small possibility of error (in producing the requested sample size) allows for an efficient and embarrassingly parallel implementation [43].

Bernoulli sampling without scaling — Given a sampling ratio P ($0<P\leq 1$), draw a random number r in $[0, 1]$ for every tuple T in D; include T in S if $r \leq P$. This is the classical formulation of Bernoulli sampling, which is rarely used in AQP systems (see below).

Bernoulli sampling with scaling — Given a sampling ratio P ($0\leq P\leq 1$), draw a random number r in $(0, 1]$ for every tuple T in D; include $\frac{1}{P}T$ in S if $r \leq P$, and otherwise add a $\mathbf{0}$ to S. Here, $\frac{1}{P}T$ is tuple T with all its attributes multiplied by $\frac{1}{P}$, and $\mathbf{0}$ is a tuple with 0 for all attributes. Thus, the sample size is the same as the original data, i.e., $|S| = |D| = N$. Despite this conceptually large sample size, this sampling technique has the same performance advantage as Bernoulli sampling without scaling since the $\mathbf{0}$ tuples are not necessarily stored and evaluated during query evaluation. The main advantage of Bernoulli sampling with scaling over Bernoulli sampling without scaling is that the sample size is no longer a random variable [51]. This greatly simplifies the derivation of the closed form error estimation (see Table 2). This type of sampling can be easily implemented for streaming or distributed environments (e.g., load shedding in a data stream management system [60]).

Stratified sampling — Given L strata $D_1, \cdots, D_L$, sample S_i of size n_i is chosen from each stratum D_i *independently*, using fixed-size sampling without replacement.[5] Our sample S is defined as $S = \cup_{i=1}^{L} S_i$, and we have $n = |S| = \cup_{i=1}^{L} n_i$.

The main advantage of stratified sampling is the potential reduction of variance, especially when the tuples are similar within each stratum but dissimilar across different strata. In AQP systems, stratified sampling has another (even more important) advantage. If the tuples are stratified based on the value of their attributes with a skewed distribution, then queries will be guaranteed to match at least a minimum number of tuples, even if their filtering conditions match rare values of those columns. This is different than classical survey sampling. In an AQP system, the effective population changes depending on the filtering conditions of the query (see Table 2 in Section 4.3).

While various sampling strategies differ in their ease of updates and amenability to parallelism, they also

[5] In this article, we only consider fixed-size sampling without replacement per stratum. However, combining the error of multiple strata follows the same principle regardless of the sampling strategy applied to each stratum.

widely differ in their approximation error and ease of closed-form estimation. In the next section, we will present the closed-form estimations of bias and variance for simple aggregate queries under each of these sampling strategies.

4.3 Closed-form Error

Our main goal in this section is to provide practitioners (and non-statisticians) a concise and accessible list of closed-form estimators for common forms of aggregates and sampling strategies.[6]

The sample estimators are widely discussed in the statistics literature. However, deriving and applying such estimators in an AQP context requires additional care. This is because database queries often have a `WHERE` clause, which effectively changes the population D and the sample S for every new query. Thus, the population size of tuples satisfying condition c can no longer be known in advance. Although the fraction of tuples in D satisfying c, F_c, can be computed by evaluating all tuples in D, such an approach is too costly. To enable fast computation, only a sample S of D must be accessed during query evaluation. For example, the fraction of tuples in S satisfying c, f_c, can be used as an approximation of F_c. However, f_c, will no longer be a constant but a random variable, and its variance must be taken into account when estimating the variance of aggregate queries. Even worse, when $f_c = 0$, one must still consider the possibility that $F_c \neq 0$. These challenges translate to more complicated closed-forms for aggregate queries with a `WHERE` clause.

In the following discussion, we divide different statistics into *computable* and *incomputable* to distinguish them based on whether they can be computed efficiently. We call a statistic or estimator *computable* if it can be computed from the original population without knowing the condition c in the query's `WHERE` clause, or from the sample. For example, N, n, and $\sum_{i=1}^{n} I_c(Y_i)$ are all computable statistics. Although the latter depends on c, it can be computed from the sample and hence, is computable. Likewise, $\gamma_c = \frac{N}{n}\sum_{i=1}^{n} I_c(Y_i)$ is also a computable estimator, as it only requires computable statistics. In contrast, we call a statistic or estimator *incomputable* if it requires certain statistics of the original population that depend on the query's `WHERE` clause condition c, or from the sample. For example, $\bar{X}_c$ and σ_{D_c} are incomputable statistics.

For ease of reference, we have summarized all our notation in Table 1. In this table, the target attribute refers to the attribute being aggregated on by the query. Table 2 provides a concise list of various estimators for simple aggregate queries under different sampling strategies.[7] The results of Table 2 can be easily generalized to `SELECT` clauses with multiple aggregates or aggregates over arithmetic expressions of the original attributes. Although omitted from Table 2, supporting the `GROUP BY` clause is also straightforward. Conceptually, each query can be re-evaluated once per every group g, each time with a new predicate c_g added its `WHERE` clause, such that c_g evaluates to true only for tuples in group g. Finally, the condition c in Table 2 is any condition that can be evaluated for each tuple in D without accessing other tuples in D. In other words, c can be either a simple condition on various attributes of tuple $T \in D$, or a nested query that does not reference D.

In Table 2, for each pair of aggregate and sampling type, we have provided the following:

1. A computable (and typically unbiased) *query estimator* for computing the approximate answer. For example, to approximate a conditional count query under fixed-sized sampling without replacement, γ_c can be computed using only the number of tuples in S satisfying c (i.e., $\sum_{i=1}^{n} I_c(Y_i)$, the size of the sample n, and the total size of the original population N.

2. The *exact expression of the query estimator's standard deviation*. However, this expression is often incomputable.

3. A computable *unbiased estimator for the query estimator's variance*, when possible. This estimator is important because the exact expression of the query estimator's standard deviation (e.g., σ_{γ_c}) is incom-

[6] The error estimation of more complex queries requires (analytical or Monte-Carlo) bootstrap, which will be discussed in Section 6.2.

[7] The references, proofs and derivations of these closed-form estimations are deferred to our technical report [49].

putable (see above).[8] When such an estimator is not available, we have instead provided computable unbiased estimators (under the right-most column of Table 2) for each of the incomputable statistics used in the expression of the query estimator's standard deviation. By substituting such statistics with their computable unbiased estimators, one can compute a reasonable (but biased) approximation of the query estimator's standard deviation.

To use the formulas presented in this article, one must use the following definitions of population and sample standard deviation:[9].

$$\sigma_D = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - \bar{X})^2} \quad \text{and} \quad \sigma_S = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$$

While in this section we focused on traditional error metrics (i.e., bias and variance), in the next section we discuss other (subtle) forms of error that can occur in an AQP system.

5 Subtle Types of Error: Subset and Superset Errors

Consider a IRS (Internal Revenue Service) table storing the income and age of different citizens living in different parts of the country:

```
IRS (citizen_name, city, age, income)
```

Now consider the following query against this table:

```
SELECT city, avg(income) as avg_income
FROM IRS
GROUP BY city
```

Let us assume that the true answer (i.e., the answer of running this query against the original table) is as follows.

Now assume that the query above is evaluated against only a sample of the original table. Then, the output aggregate values will be approximate. For example:

Here $\pm$ represents a confidence interval at a certain level of confidence, e.g., 95%. The problem here is that if a city (e.g., Ann Arbor) has significantly fewer tuples in the original table than larger cities (e.g., New York), then it may not be present in the selected sample. Thus, using such a sample will naturally leave out the group pertaining to Ann Arbor from the query's output altogether. When this situation occurs, the error is no longer one of *aggregation error*. In other words, Ann Arbor will not be even present in the output to be accompanied with accuracy guarantees for its aggregate value (here, average income). This phenomenon has been the motivation behind the wide use of stratified samples, increasing the likelihood of rare groups being present in the sample [8, 20, 11, 66]. Even when a reliable estimate cannot be obtained for a particular group, one would wish to at least know that such a group should have existed. In other words, the following would be an ideal output:

Unfortunately, the use of stratified samples does not always prevent missing records from the output. This problem, called *subset error*, becomes more evident by considering the following slightly more involved query.

[8]The square root of our unbiased estimators for variance may be used as an approximation of the standard deviation. However, note that the square root of an unbiased estimator for a random variable is not necessarily an unbiased estimator for the square root of that random variable!

[9]In other words, we do *not* use the $\sigma_S = \sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$ definition. Instead, we apply a correction factor separately.

Symbol	Meaning
D	The entire set of tuples
S	The selected sample of tuples, i.e., $S \subseteq D$
N	The number of original tuples, i.e., $N = \|D\|$
n	The number of tuples in the sample, i.e., $n = \|S\|$
c	The condition used in the `WHERE` clause
I_c	The indicator function, where $I_c(t) = 1$ if tuple t satisfies c and $I_c(t) = 0$ otherwise
F_c	The fraction of tuples in D satisfying condition c $(0 \leq F_c \leq 1)$
f_c	The fraction of tuples in S satisfying condition c $(0 \leq f_c \leq 1)$
N_c	The number of tuples in S satisfying condition c $(0 \leq N_c \leq N)$
$T_1, \cdots, T_N$	The tuples in D
$t_1, \cdots, t_n$	The tuples in S
$X_1, \cdots, X_N$	The values of the target attribute in $T_1, \cdots, T_N$
$Y_1, \cdots, Y_n$	The values of the target attribute in $t_1, \cdots, t_n$
$Z_{c_1}, \cdots, Z_{c_N}$	$Z_{c_i} = X_i$ if $I_c(T_i) = 1$ and $Z_{c_i} = 0$ otherwise
$\bar{X}$	The mean of the target attribute in D, i.e., $\bar{X} = \frac{1}{N}\sum_{i=1}^{N} X_i$
$\bar{Y}$	The mean of the target attribute in S, i.e., $\bar{Y} = \frac{1}{n}\sum_{i=1}^{n} Y_i$
$\bar{X}_c$	The conditional mean of the target attribute in D, i.e., $X_c = \frac{\sum_{i=1}^{N} X_i I_c(X_i)}{\sum_{i=1}^{N} I_c(X_i)}$
$\bar{Y}_c$	The conditional mean of the target attribute in S, i.e., $Y_c = \frac{\sum_{i=1}^{N} Y_i I_c(Y_i)}{\sum_{i=1}^{N} I_c(Y_i)}$
$\bar{Z}_c$	The mean of the Z_{c_i} values, i.e., $\bar{Z}_c = \frac{1}{N}\sum_{i=1}^{N} Z_{c_i}$
σ_D	The standard deviation of the target attribute in D, i.e., $\sigma_D = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(X_i - \bar{X})^2}$
σ_S	The standard deviation of the target attribute in S, i.e., $\sigma_S = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(Y_i - \bar{Y})^2}$
σ_Z	The standard deviation of the Z_{c_i} values, i.e., $\sigma_Z = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(Z_{c_i} - \bar{Z}_c)^2}$
σ_{ZY}	The standard deviation of the Z_{c_i} values for items included in S
σ_{ZY_i}	The standard deviation of the Z_{c_i} values for items included in S_i
σ_{D_c}	The standard deviation of the target attribute in $\{X_i \mid T_i \in D,\ I_c(T_i) = 1\}$
σ_{S_c}	The standard deviation of the target attribute in $\{Y_i \mid t_i \in S,\ I_c(t_i) = 1\}$
P	The sampling probability in Bernoulli sampling $(0 < P \leq 1)$
L	The number of strata in stratified sampling $(1 \leq L \leq N)$.
$D_1, \cdots, D_L$	The strata of D, i.e., $D = \cup_{i=1}^{L} D_i$ and $D_i \cap D_j = \emptyset$ for $i \neq j$
$S_1, \cdots, S_L$	The samples from different strata of D, i.e., $S = \cup_{i=1}^{L} S_i$ and $S_i \subseteq D_i$
$N_1, \cdots, N_L$	The number of tuples in different strata, i.e., $N_i = \|D_i\|$ and $N = \sum_{i=1}^{L} N_i$
$N_{c_1}, \cdots, N_{c_L}$	The number of tuples in different strata satisfying c, i.e., $0 \leq N_{c_i} \leq N_i$
$n_1, \cdots, n_L$	The number of sampled items from each stratum, i.e., $n_i = \|S_i\|$ and $n = \sum_{i=1}^{L} n_i$
$\bar{X}_1, \cdots, \bar{X}_L$	The mean of the target attribute in each stratum
$\bar{Y}_1, \cdots, \bar{Y}_L$	The mean of the target attribute in the sample from each stratum
$\bar{Z}_1, \cdots, \bar{Z}_L$	The mean of the Z_{c_i} values in each stratum
$F_{c_1}, \cdots, F_{c_L}$	The fraction of tuples satisfying c in each stratum
$f_{c_1}, \cdots, f_{c_L}$	The fraction of tuples satisfying c in each S_i
σ_{D_i}	The standard deviation of the target attribute in D_i
σ_{S_i}	The standard deviation of the target attribute in S_i
$\sigma_{D_{c_i}}$	The standard deviation of the target attribute for those tuples in D_i that satisfy c
$\sigma_{S_{c_i}}$	The standard deviation of the target attribute for those tuples in S_i that satisfy c
σ_{Z_i}	The standard deviation of the Z_{c_i} values in D_i
θ	The estimator for approximating $\bar{X}$ using S $(\texttt{AVG}(X)\ \texttt{FROM}\ D)$
ϕ	The estimator for approximating $\sum_{i=1}^{N} X_i$ using S $(\texttt{SUM}(X)\ \texttt{FROM}\ D)$
γ_c	The estimator for approximating $\sum_{i=1}^{N} I_c(X_i)$ using S $(\texttt{COUNT} * \texttt{FROM}\ D\ \texttt{WHERE}\ c)$
θ_c	The estimator for approximating $\bar{X}_c$ using S $(\texttt{AVG}(X)\ \texttt{FROM}\ D\ \texttt{WHERE}\ c)$
ϕ_c	The estimator for approximating $\sum_{i=1}^{N} X_i I_c(X_i)$ using S $(\texttt{SUM}(X)\ \texttt{FROM}\ D\ \texttt{WHERE}\ c)$
σ_θ	The standard deviation of θ
σ_ϕ	The standard deviation of ϕ
σ_{γ_c}	The standard deviation of γ_c
σ_{θ_c}	The standard deviation of θ_c
σ_{ϕ_c}	The standard deviation of ϕ_c

Table 1: Notation used in this paper.

Sampling Strategy	**Mean Queries:** `SELECT AVG(X) FROM D`	**Conditional Mean Queries:** `SELECT AVG(X) FROM D WHERE c`	**Conditional Sum Queries:** `SELECT SUM(X) FROM D WHERE c`	**Conditional Count Queries:** `SELECT COUNT(*) FROM D WHERE c`	**Other sample-based estimators for population-based quantities**
Fixed-size with Replacement	$\theta = \bar{Y}$ (unbiased) $\sigma_\theta = \frac{\sigma_D}{\sqrt{n}}$ $\sigma_\theta^2 = \mathbb{E}\left[\frac{1}{n-1}\sigma_S^2\right]$	$\theta_c = \bar{Y}_c$ (biased: $\mathbb{E}[\theta_c] = (1-(1-F_c)^n)\bar{X}_c$) $\sigma_{\theta_c} = \sqrt{\sum_{k=1}^{n} \frac{\sigma_{D_c}^2}{k}\binom{n}{k}F_c^k(1-F_c)^{n-k} + \bar{X}_c^2(1-(1-F_c)^n)(1-F_c)^n}$ (no computable unbiased estimator for $\sigma_{\theta_c}^2$ unless $F_c = 1$)	$\phi_c = \frac{N}{n}\sum_{i=1}^{n} Y_i I_c(Y_i)$ (unbiased) σ_{ϕ_c}=$N\frac{\sigma_Z}{\sqrt{n}}$ $\sigma_{\phi_c}^2 = \mathbb{E}\left[\frac{N^2}{n-1}\sigma_{ZY}^2\right]$	$\gamma_c = N f_c$(unbiased) $\sigma_{\gamma_c} = N\sqrt{\frac{F_c(1-F_c)}{n}}$ $\sigma_{\gamma_c}^2 = \mathbb{E}\left[\frac{N^2}{n-1}f_c(1-f_c)\right]$	$\sigma_{D_c}^2 = \mathbb{E}\left[\frac{1}{M}\sigma_{S_c}^2\right]$ $M = \sum_{k=1}^{n}\frac{k-1}{k}\binom{n}{k}F_c^k(1-F_c)^{n-k}$ $F_c = \mathbb{E}[f_c]$
Fixed-size without Replacement	$\theta = \bar{Y}$ (unbiased) $\sigma_\theta = \sigma_D\sqrt{\frac{N-n}{n(N-1)}}$ $\sigma_\theta^2 = \mathbb{E}\left[\frac{N-n}{N(n-1)}\sigma_S^2\right]$	$\theta_c = \bar{Y}_c$ $\mathbb{E}[\theta_c] = \bar{X}_c W\begin{cases} biased & if\ n \leqslant N - N_c \\ unbiased & if\ n > N - N_c \end{cases}$ $\sigma_{\theta_c} = \sqrt{\sigma_{D_c}^2\frac{N_c}{N_c-1}H_c + \bar{X_c}^2(1-W)W}$ $a = max\{1, n-N+N_c\},\ b = min\{n, N_c\},$ $W = \begin{cases} \sum_{k=a}^{b}\frac{\binom{N_c}{k}\binom{N-N_c}{n-k}}{\binom{N}{n}} & n \leqslant N - N_c \\ 1 & n > N - N_c \end{cases}$ $H_c = \sum_{k=a}^{b}\left(\frac{1}{k}-\frac{1}{N_c}\right)\frac{\binom{N_c}{k}\binom{N-N_c}{n-k}}{\binom{N}{n}}$ (no computable unbiased estimator for $\sigma_{\theta_c}^2$ unless $F_c = 1$)	$\phi_c = \frac{N}{n}\sum_{i=1}^{n} Y_i I_c(Y_i)$ (unbiased) σ_{ϕ_c}=$N\sigma_Z\sqrt{\frac{N-n}{n(N-1)}}$ $\sigma_{\phi_c}^2 = \mathbb{E}\left[\frac{N(N-n)}{n-1}\sigma_{ZY}^2\right]$	$\gamma_c = N f_c$ (unbiased) $\sigma_{\gamma_c} = N\sqrt{\frac{N-n}{n(N-1)}F_c(1-F_c)}$ $\sigma_{\gamma_c}^2 = \mathbb{E}\left[\frac{N(N-n)}{n-1}f_c(1-f_c)\right]$	$\sigma_{D_c}^2 = \frac{1}{M}\mathbb{E}\left[\sigma_{S_c}^2\right]$ $M = \frac{N_c}{N_c-1}\sum_{k=a}^{b}\frac{k-1}{k}\frac{\binom{N_c}{k}\binom{N-N_c}{n-k}}{\binom{N}{n}}$ $F_c = \mathbb{E}[f_c]$
Bernoulli without Scaling	$\theta = \bar{Y}$ (biased $E[\theta] = (1-(1-P)^N)\bar{X}$) $\sigma_\theta = \sqrt{\frac{N}{N-1}\sigma_D^2 H + (1-P)^N(1-(1-P)^N)\bar{X}^2}$ H=$\sum_{k=1}^{N}(\frac{1}{k})\binom{N}{k}P^k(1-P)^{N-k} - \frac{(1-(1-P)^N)}{N}$	$\theta_c = \bar{Y}_c$ (biased $E[\theta_c] = (1-(1-P)^{N_c})\bar{X}_c$) $\sigma_{\theta_c} = \sqrt{\frac{N_c}{N_c-1}\sigma_{D_c}^2 H_c + (1-P)^{N_c}(1-(1-P)^{N_c})\bar{X_c}^2}$ H_c=$\sum_{k=1}^{N_c}(\frac{1}{k})\binom{N_c}{k}P^k(1-P)^{N_c-k} - \frac{(1-(1-P)^{N_c})}{N_c}$	$\phi_c = \frac{N}{n}\sum_{i=1}^{n} Y_i I_c(Y_i)$ (biased ,$\mathbb{E}[\phi_c] = (1-(1-P)^N)N\bar{Z})$ σ_{ϕ_c}=$N\sqrt{\frac{N}{N-1}\sigma_Z^2 H + (1-P)^N(1-(1-P)^N)\bar{Z}^2}$ $H = \sum_{k=1}^{N}(\frac{1}{k})\binom{N}{k}P^k(1-P)^{N-k} - \frac{(1-(1-P)^N)}{N}$	$\gamma_c = N f_c$ (biased $\mathbb{E}[\gamma_c] = (1-(1-P)^N)N_c$) $\sigma_{\gamma_c} = N\sqrt{F_c(1-F_c)\frac{N}{N-1}H + (1-P)^N(1-(1-P)^N)F_c^2}$ $H = \sum_{k=1}^{N}(\frac{1}{k})\binom{N}{k}P^k(1-P)^{N-k} - \frac{(1-(1-P)^N)}{N}$	$\sigma_{D_c}^2 = \frac{1}{M}\mathbb{E}\left[\sigma_{S_c}^2\right]$ $M = \sum_{k=1}^{N_c}\frac{N_c(k-1)}{k(N_c-1)}\binom{N_c}{k}P^k(1-P)^{N_c-k}$ $F_c = \mathbb{E}\left[\frac{f_c}{1-(1-P)^N}\right]$
Bernoulli with Scaling	$\theta = \frac{1}{N}\sum_{i=1}^{n} Y_i/P$ (unbiased) $\sigma_\theta = \sqrt{\frac{1-P}{NP}(\sigma_D^2 + \bar{X}^2)}$ $\sigma_\theta^2 = \mathbb{E}\left[\frac{1-P}{N^2}\sum_{i=1}^{n} Y_i^2\right]$	$\theta_c = \frac{\sum_{i=1}^{n} Y_i I_c(Y_i)/P}{\sum_{i=1}^{n} I_c(Y_i)/P} = \bar{Y}_c$ (biased, $E[\theta_c] = (1-(1-P)^{N_c})\bar{X}_c$) $\sigma_{\theta_c} = \sqrt{\frac{N_c}{N_c-1}\sigma_{D_c}^2 H_c + (1-P)^{N_c}(1-(1-P)^{N_c})\bar{X_c}^2}$ H_c=$\sum_{k=1}^{N_c}(\frac{1}{k})\binom{N_c}{k}P^k(1-P)^{N_c-k} - \frac{(1-(1-P)^{N_c})}{N_c}$	$\phi_c = \sum_{i=1}^{n} Y_i I_c(Y_i)/P$ (unbiased) σ_{ϕ_c}=$\sqrt{\frac{1-P}{P}N(\sigma_Z^2 + \bar{Z}^2)}$ $\sigma_{\phi_c}^2 = \mathbb{E}\left[(1-P)\sum_{i=1}^{n} Y_i I_c(Y_i)^2\right]$	$\gamma_c = \sum_{i=1}^{n} I_c(Y_i)/P$ (unbiased) $\sigma_{\gamma_c} = \sqrt{\frac{1-P}{P}NF_c}$ $\sigma_{\gamma_c}^2 = \mathbb{E}\left[\frac{1-P}{P}Nf_c\right]$	$\sigma_{D_c}^2 = \frac{1}{M}\mathbb{E}\left[\sigma_{S_c}^2\right]$ $M = \sum_{k=1}^{N_c}\frac{N_c(k-1)}{k(N_c-1)}\binom{N_c}{k}P^k(1-P)^{N_c-k}$ $F_c = \mathbb{E}[f_c]$
Stratified Sampling	$\theta = \frac{1}{N}\sum_{i=1}^{L} N_i\bar{Y}_i$ (unbiased) $\sigma_\theta = \frac{1}{N}\sqrt{\sum_{i=1}^{L}\frac{(N_i-n_i)}{n_i(N_i-1)}N_i^2\sigma_{Di}^2}$ $\sigma_\theta^2 = \mathbb{E}\left[\frac{1}{N^2}\sum_{i=1}^{L}\frac{N_i(N_i-n_i)}{n_i-1}\sigma_{S_i}^2\right]$	$\theta_c = \frac{1}{N}\sum_{i=1}^{L} N_i\frac{\sum_{Y_j\in S_i} Y_j I_c(Y_j)}{\sum_{Y_j\in S_i} I_c(Y_j)}$ (biased) $\mathbb{E}[\theta_c] = \frac{1}{N}\sum_{i=1}^{L} N_i\bar{X}_{c_i}W_i$ $\sigma_{\theta_c} = \frac{1}{N}\sqrt{\sum_{i=1}^{L} N_i^2\left(\sigma_{D_{c_i}}^2\frac{N_{c_i}}{N_{c_i}-1}H_{c_i} + \bar{X_{c_i}}^2(1-W_i)W_i\right)}$ $H_{c_i} = \sum_{k=a_i}^{b_i}\left(\frac{1}{k}-\frac{1}{N_{c_i}}\right)\frac{\binom{N_{c_i}}{k}\binom{N_i-N_{c_i}}{n_i-k}}{\binom{N_i}{n_i}}$ $W_i = \begin{cases} \sum_{k=a_i}^{b_i}\frac{\binom{N_{c_i}}{k}\binom{N_i-N_{c_i}}{n_i-k}}{\binom{N_i}{n_i}} & n_i \leqslant N_i - N_{c_i} \\ 1 & n_i > N_i - N_{c_i} \end{cases}$ $a_i = max\{1, n_i - N_i + N_{c_i}\}, b_i = min\{n_i, N_{c_i}\}$ (no computable unbiased estimator for $\sigma_{\theta_c}^2$ unless $F_c = 1$)	$\phi_c = \sum_{i=1}^{L} N_i\bar{Z}_i$ (unbiased) σ_{ϕ_c}=$\sqrt{\sum_{i=1}^{L}\frac{(N_i-n_i)}{n_i(N_i-1)}N_i^2\sigma_{Zi}^2}$ $\sigma_{\phi_c}^2 = \mathbb{E}\left[\sum_{i=1}^{L}\frac{N_i-n_i}{N_i(n_i-1)}\sigma_{ZY_i}^2\right]$	$\gamma_c = \sum_{i=1}^{L}\frac{N_i}{n_i}\sum_{Y_j\in S_i} I_c(Y_j)$ (unbiased) $\sigma_{\gamma_c} = \sqrt{\sum_{i=1}^{L}\frac{(N_i-n_i)}{n_i(N_i-1)}N_i^2 F_{c_i}(1-F_{c_i})}$ $\sigma_{\gamma_c}^2 = \mathbb{E}\left[\sum_{i=1}^{L}\frac{N_i(N_i-n_i)}{(n_i-1)}f_c(1-f_c)\right]$	$\sigma_{D_{c_i}}^2 = \frac{1}{M}\mathbb{E}\left[\sigma_{S_{c_i}}^2\right]$ $M = \frac{N_{c_i}}{N_{c_i}-1}\sum_{k=a_i}^{b_i}\frac{k-1}{k}\frac{\binom{N_{c_i}}{k}\binom{N_i-N_{c_i}}{n_i-k}}{\binom{N_i}{n_i}}$ $F_{c_i} = \mathbb{E}[f_{c_i}]$ $N_{c_i} = \mathbb{E}[N_i f_{c_i}]$ $\bar{X}_{c_i} = \mathbb{E}\left[\frac{\bar{Y}_{c_i}}{\sum_{k=a_i}^{b_i}\frac{\binom{N_{c_i}}{k}\binom{N_i-N_{c_i}}{n_i-k}}{\binom{N_i}{n_i}}}\right]$

Table 2: The error estimation results for different aggregate functions under different sampling strategies.

city	avg_income
New York	13,010
Los Angeles	34,560
Detroit	12,800
Ann Arbor	5,000

city	avg_income
New York	12,600 $\pm$ 700
Los Angeles	33,200 $\pm$ 1,100
Detroit	13,100 $\pm$250

```
SELECT city, avg(income) as avg_income
FROM IRS
GROUP BY city
HAVING avg(income) > 13,000
```

The true answer to this query would be the following: However, using the same sample as the previous query, the approximate answer returned for this query will be as follows:

The problem with this approximation is self-evident. Certain tuples are missing from the output altogether, i.e., the row pertaining to New York city would have been present in the output of the query if executed against the original data. This is called *subset error*. This error cannot be easily prevented with stratified samples. Here, for example, the reason for New York's row being missing from the output is that our current estimate of its average income is $12,600$ which is below the $13,000$ cut-off point set by the HAVING clause.

Likewise, certain tuples should not have been present in the output of this query (i.e., the row pertaining to Detroit) as its true aggregate value in reality is $12,800$ which is lower than $13,000$. However, we have a group for Detroit in our output simply because our current estimate of Detroit's average income happens to be $13,100$. This is the opposite side of the subset error, and is called the *superset error*.

Confidence intervals and other range-based accuracy guarantees are only apt at expressing the error of point estimates, such as the error of an aggregate function's output. However, as evident from these simple examples, subset and superset errors are of a different nature. These types of errors originate from the relational operators (e.g., selection, grouping, and projection) and are almost unheard of in the statistics literature. However, both subset and superset errors can be easily defined using the possible worlds semantics widely used in probabilistic query processing literature [23]. (See [67] as an example of the close connection between probabilistic and approximate databases). In particular, the subset and superset errors could be managed by computing the *probability of existence* for each possible output group [67]. For example, the following output will be far more informative (and semantically more consistent) than the previous output:

This additional probability accompanying each output tuple can easily cause further confusion and lead to an overwhelming number of output tuples. However, the purpose of this section is to simply point out the need for AQP systems to compute and treat these probabilities in a principled manner, in order to ensure correct semantics. Later, in Section 7, we discuss several approaches to hide this added complexity from the end users.

6 An Important Error Often Overlooked: Bias

As mentioned in Section 4, it is important to provide *unbiased estimates* to ensure that in the long run all the approximate answers are centered around the true answer. First, in Section 6.1, we show that the plug-in

city	avg_income
New York	12,600 ± 700
Los Angeles	33,200 ± 1,100
Detroit	13,100 ±250
Ann Arbor	NaN ± ∞

city	avg_income
New York	13,010
Los Angeles	34,560

approach can produce biased estimates even for relatively simple queries. Then, in Sections 6.3 and 6.4, we propose the use of bootstrap for estimating and correcting the bias.

6.1 Bias in the Plug-in Approach

Here, we show how the plug-in approach can lead to statistical bias even for queries that are relatively simple. Consider T3 as an instance of the IRS relation (shown in Table 3). Assume that the user wants to execute Q2 against T3. The true answer to this query will therefore be $\theta = Q_2(T_3) = (100K+200K+250K)/3 = 183.3K$.

For the sake of this toy example, let us assume that the AQP system decides to use the plug-in approach on a sample of size 3 to answer this query, i.e., $|S| = 3$. To compute the expected value of the returned approximation, $\mathbb{E}_{|S|=k}[\hat{\theta}_S]$, we need to consider all possible samples of size 3. Since $\binom{4}{3} = 4$, we can enumerate all these possibilities for this toy example:

$$\mathbb{E}_{|S|=3}(\theta(S)) = 100K \cdot (5/3 + 2 + 7/4 + 9/4)/4 = 191.7K$$

Thus, the bias b of our estimator for Q_2 is non-zero:

$$b = \mathbb{E}_{|S|=3}(\theta(S)) - \theta = (191.7 - 183.3) * 1000 = 8.4K \neq 0 \quad (3)$$

Note that this query was quite simple: it only consisted of two AVG aggregates, with no HAVING or even WHERE clause. Yet, the plug-in approach led us to a biased answer.[10] Almost all existing AQP systems have assumed (at least, implicitly) that the execution of the original query on a sample (i.e., the plug-in approach) yields an unbiased estimate of the true answer. Consequently, to this date, all error quantification efforts in the AQP community have focused on estimating the variance of the approximate answers. Given that the sampling error is often asymptotically normal, the estimated variance can be used to provide a confidence interval centered around the approximate answer. This is based on the assumption that the approximate answer from the plug-in approach is an unbiased estimate. Even more recent studies using bootstrap for error quantification assume that an unbiased estimate (e.g., in form of a re-written query) is provided by the user [67]. In other words, these techniques have only been used for estimating the variance but not the bias of the estimators.

In the next section, we explain how (non-parametric) bootstrap can be used to estimate the bias of the approximate answer produced by the plug-in approach. Once estimated, the bias can be subtracted from the original estimate to produce an unbiased approximate answer. For the unfamiliar reader, we first provide a brief introduction to the bootstrap theory.

[10] As an example of an even simpler query (i.e., with no nested sub-query) that still leads to bias, consider the following query: `SELECT AVG(X) FROM D WHERE X > 3` that is to be run against a sample of size $n = 1$ chosen from a table $D = \{0, 5\}$. There are two possible samples $\{0\}$ and $\{5\}$, with equal probabilities. These samples yield 0 and 5, respectively, as an approximate answer. Thus, the expected value of our approximation will be $\frac{0+5}{2} = 2.5$, while the true answer is 5.

city	avg_income
Los Angeles	33,200 ± 1,100
Detroit	13,100 ± 250

city	avg_income	probability_of_existence
New York	12,600 ± 700	0.98
Los Angeles	33,200 ± 1,100	0.999999
Detroit	13,100 ± 250	0.90
Ann Arbor	4,010 ± 150	0.0000001

6.2 Background: Bootstrap

In this section, we provide a brief overview of the *nonparametric bootstrap* (or simply the *bootstrap*), which has many applications in AQP systems. Bootstrap [26] is a powerful statistical technique traditionally designed for estimating the uncertainty of sample estimators. Consider an estimator θ (i.e., a query) that is evaluated on a sample S. This estimated value, denoted as $\theta(S)$, is a point-estimate (i.e., a single value), and hence, reveals little information about how this value would have changed had we used a different sample S'. This information is critical as the answer could be quite different from one sample to the next. Thus, S should be treated as a random variable drawn from the original population (dataset D in our case). Consequently, statisticians are often interested in measuring *distributional information* about $\theta(S)$, such as variance, bias, etc. Ideally, one could measure such statistics by (i) drawing many new samples, say $S_1, \cdots, S_k$ for some large k, from the same population D from which the original S was drawn, such that $|S| = |S_i|$, (ii) computing $\theta(S_1), \cdots, \theta(S_k)$, and finally (iii) inducing a distribution for $\theta(S)$ based on the observed values of $\theta(S_i)$. We call this the true distribution of $\theta(S)$. Figure 3a illustrates this computation. Given this distribution, any distributional information (e.g., variance) can be computed.

Unfortunately, in practice, the original dataset D cannot be easily accessed (e.g., due to its large size), and therefore, direct computation of $\theta(S)$'s distribution using the procedure of Figure 3a is impossible. This is where bootstrap [26] becomes useful. The main idea of bootstrap is simple: treat S as a proxy for its original population D. In other words, instead of drawing S_i's directly from D, generate new samples $S_1^*, \cdots, S_k^*$ by resampling from S itself. Each S_i^* is called a (bootstrap) replicate or simply a bootstrap. Each S_i^* is generated by drawing $n = |S|$ independently and identically distributed (I.I.D.) samples *with replacement* from S, and thus, some elements of S might be repeated or missing in each S_i^*. Note that all bootstraps have the same cardinality as S, i.e. $|S_i^*| = |S|$ for all i. By computing θ on these bootstraps, namely $\theta(S_1^*), \cdots, \theta(S_k^*)$, we can create an empirical distribution of $\theta(S)$. This is the bootstrap computation, which is visualized in Figure 3b.

The theory of bootstrap guarantees that for a large class of estimators θ and sufficiently large k, we can use the generated empirical distribution of $\theta(S)$ as a consistent approximation of its true distribution. The intuition is that, by resampling from S, we emulate the original population D from which S was drawn. Here, it is sufficient (but not necessary) that θ be relatively smooth (i.e., Hadamard differentiable [26]) which holds for a large class of queries.[11] In practice, $k = 100$ is usually sufficient for achieving reasonable accuracy (k can also be tuned automatically; see [26]).

Advantages — Employing bootstrap has several key advantages for AQP systems. First, as noted, bootstrap delivers consistent estimates for a large class of estimators. Second, the bootstrap computation uses a "plug-in" principle; that is, we simply need to re-evaluate our query θ on S_i^*'s instead of S. Thus, we do not need to modify or re-write θ. Finally, individual bootstrap computations $\theta(S_1^*), \cdots, \theta(S_k^*)$ are independent from each other, and hence can be executed in parallel. This embarrassingly parallel execution model enables scalability

[11] Bootstrap has even been used for estimating the uncertainty of complex functions, such as machine learning classifiers (see [50]).

citizen_name	city	age	income
John	Detroit	38	100K
Alice	Ann Arbor	35	200K
Bob	Los Angeles	25	200K
Mary	Los Angeles	42	300K

Table 3: T_3: another instance of the IRS relation.

```
SELECT AVG(city_income)
FROM (
        SELECT city, AVG(income) as city_income
        FROM IRS
        GROUP BY city
);
```

Figure 2: Q_2: a simple query, computing the average of the average income across all cities.

by taking full advantage of modern many-core and distributed systems.

Disadvantages — The downside of bootstrap is its computational overhead, as hundreds or thousands of bootstrap trials are typically needed to obtain reliable estimates. However, several optimizations have been proposed for speeding up the bootstrap process. Pol et al. [56] have proposed Tuple Augmentation (TA) and On-Demand Materialization (ODM) for efficient generation of bootstrap samples, while Laptev et al. [30] have exploited the computation overlap across different bootstrap trials. Agarwal et al. [10] have shown that leveraging Poissonized resampling (as opposed to sampling with replacement) can yield significant speed ups. An alternative approach, called *Analytical Bootstrap Method* (ABM) [66, 67], avoids the Monte Carlo simulations altogether; instead, ABM uses a probabilistic relational model for the bootstrap process to automatically estimate the error (for a large class of SQL queries) to bypass the actual Monte Carlo simulation. Unlike the Poissonized resampling, ABM is limited to SQL operators and cannot handle user-defined aggregates.

6.3 Bias Estimation and Correction Using Bootstrap

The bias b of the plug-in approach is defined as:

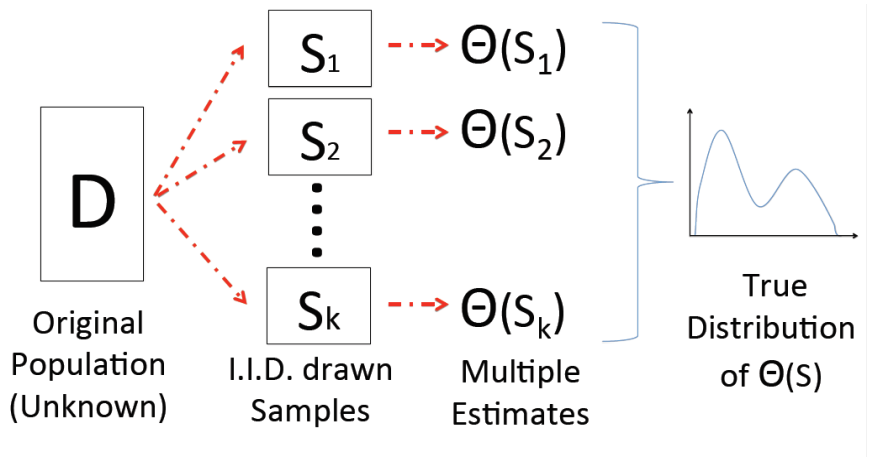
$$b = \mathbb{E}_{|S|=k}[\theta_S] - \theta = \mathbb{E}_{|S|=k}[q(S)] - q(D) \tag{4}$$

Using bootstrap theory, we can estimate $\mathbb{E}_{|S|=k}[q(S)] - q(D)$ by replacing S with S^* and D with S, as follows:

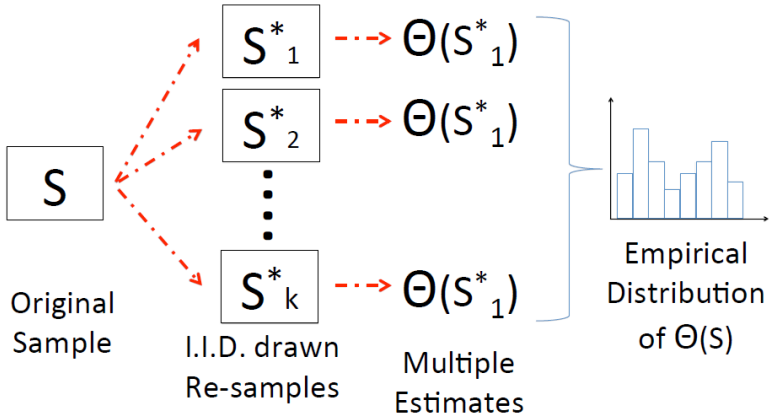
$$b \approx \mathbb{E}_{|S^*|=k}[q(S^*)] - q(S) \tag{5}$$

where S^* is a bootstrap resample of S and the expectation is taken over all bootstrap resamples. Since enumerating all possible bootstrap resamples is infeasible, we often restrict ourselves to m independent resamples of S for sufficiently large values of m. We denote these bootstrap resamples by $S_1, S_2, ..., S_B$, and take the average of $q(S_i)$ over $i \in \{1, 2, ..., m\}$:

$$b \approx \frac{1}{m} \sum_{i=1}^{m} q(S_i) - q(S) \tag{6}$$



(a) Ideal computation of $\theta(S)$'s distribution



(b) Bootstrap computation of $\theta(S)$'s distribution

Figure 3: Since the original data D may not be easily accessible to generate new samples S_i from, bootstrap uses resamples S_i^* of the original sample S instead, where $|S| = |S_i^*|$, to produce an empirical distribution for $\theta(S)$.

Choosing a large m (many bootstrap resamples) improves the accuracy of the bias estimation. The computational overhead of repeatedly evaluating q on each resample can be avoided by using the analytical bootstrap method (ABM) [67].

As a toy example, we apply this bias estimation technique to query Q2 and relation T3. Assume that $k = 3$ and that the chosen sample is $S = \{(200K, AnnArbor), (200K, LosAngeles), (300K, LosAngeles)\}$. Here, for brevity, we only display the income and city fields as the other fields do not affect the query results. For this small example, we can exhaustively enumerate all possible bootstrap resamples, their appearance probability, and their corresponding query result $q(S_i)$:

Thus, we have:

$$\mathbb{E}_{|S|=3}[q(S^*)] = 100K(2 * 8/27 + 3 * 1/27 + 2.5 * 6/27 + 7/3 * 3/27 + 8/3 * 3/27 + 9/4 * 6/27) = 231.5K$$

Since $q(S) = (200K + 250K)/2 = 225K$, bootstrap will estimate the bias as $b \approx 231.5K - 225K = 6.5K$. This value is quite close to the true bias, which as computed in (3) is $b = 8.4K$. However, while this bias estimation strategy works well for the toy example at hand, it is important to note that this technique can be highly inaccurate when the sample size (i.e., $k = |S|$) or the number of resamples (i.e., m) are small. In the next section, we explain a reentering technique that can improve the accuracy of this bias estimation strategy.

Table 4: Bootstrap resamples and their corresponding probability of appearance.

Resample	Probability of Appearance	$q(S_i)$
{(200K,Ann Arbor) , (200K,Ann Arbor) , (200K,Ann Arbor)}	1/27	200K
{(200K,Los Angeles), (200K,Los Angeles), (200K,Los Angeles)}	1/27	200K
{(300K,Los Angeles), (300K,Los Angeles), (300K,Los Angeles)}	1/27	300K
{(200K,Ann Arbor) , (200K,Los Angeles), (200K,Los Angeles)}	3/27	200K
{(200K,Ann Arbor) , (200K,Ann Arbor) , (200K,Los Angeles)}	3/27	200K
{(200K,Ann Arbor) , (200K,Ann Arbor) , (300K,Los Angeles)}	3/27	500K/2
{(200K,Ann Arbor) , (300K,Los Angeles), (300K,Los Angeles)}	3/27	500K/2
{(200K,Los Angeles), (200K,Los Angeles), (300K,Los Angeles)}	3/27	700K/3
{(200K,Los Angeles), (300K,Los Angeles), (300K,Los Angeles)}	3/27	800K/3
{(200K,Ann Arbor) , (200K,Los Angeles), (300K,Los Angeles)}	6/27	900K/4

6.4 Bias Estimation and Correction Using Bootstrap and Recentering

Let $S = \{X_1, X_2, ..., X_k\}$, where each X_j is a tuple. Also, let $S_1, S_2, ..., S_m$ be m bootstrap resamples of S. To apply the recentering method, we represent each S_i by a vector $V_i = (a_{i,1}, a_{i,2}, \cdots, a_{i,k})$, where $a_{i,j}$ is the number of times that tuple X_j is repeated in S_i. Thus, based on the definition of bootstrap resamples, $\sum_{j=1}^{k} a_{i,j} = k$. For example, if S_1 consists of $k-1$ repetitions of X_1 and one instance of X_k, then $V_1 = (k-1, 0, \cdots, 0, 1)$. We abuse our notation to use this vector representation: we replace $q(S_i)$ with $q(V_i)$. Let $\bar{V}$ denote the average of all V_i's:

$$\bar{V} = \frac{1}{m}\sum_{i=1}^{m} V_i$$

With this notation, the bootstrap with recentering technique [26] estimates the bias of an approximation as follows:

$$b \approx \frac{1}{m}\sum_{i=1}^{m} q(S_i) - q(\bar{V}) \tag{7}$$

The difference between equations (6) and (7) is that, in the latter, we have replaced $q(S)$ with $q(\bar{V})$.

The intuition behind why (7) is superior in accuracy to (6) is as follows. Note that $\frac{1}{m}\sum_{i=1}^{m} q(S_i)$ in (6) is used as an approximation of $\mathbb{E}_{|S|=k}[q(S^*)]$ in equation (5). However, unless we consider all (or many) possible resamples of S, this approximation can be quite inaccurate. Due to computational constraints, m may not be sufficiently large in practice. To compensate for this shortcoming, the recentering technique (conceptually) changes the distribution of the tuples in S to enforce that $\frac{1}{m}\sum_{i=1}^{m} q(S_i)$ is an accurate approximation of $\mathbb{E}_{|S|=k}[q(S^*)]$. In other words, before estimating the bias, we replace S with a new sample that fits the empirical distribution formed by our m bootstrap resamples. This new sample is our $\bar{V}$, which is formed by averaging the m bootstrapped resamples. It has been shown that this reentering strategy exhibits less variance than the technique described in Section 6.3 [26].

The toy example used in Section 6.3 enumerated all bootstrap resamples. When this is the case, it is easy to show that $q(S) = q(\bar{V})$, which causes the two techniques to produce identical estimates of the bias. However, in practice, enumerating all possible resamples is often infeasible, and thus, the reentering technique becomes superior. To demonstrate this, we reconsider the same query Q2 and sample S as used in the previous section,

but limit ourselves to only $m = 6$ bootstrap resamples $S_1, S_2, \cdots, S_6$. These resamples, their V_i representation, and their corresponding $q(S_i)$ are all listed in the following table:

Table 5: Bootstrap resamples and their vector representations

S_i	V_i	$q(S_i)$
S_1 ={(200K,Ann Arbor) ,(300K,Los Angeles),(300K,Los Angeles)}	$V_1 = (1, 0, 2)$	$q(S^1) = 500K/2$
S_2 ={(300K,Los Angeles),(300K,Los Angeles),(300K,Los Angeles)}	$V_2 = (0, 0, 3)$	$q(S^2) = 300K$
S_3 ={(200K,Los Angeles),(300K,Los Angeles),(300K,Los Angeles)}	$V_3 = (0, 1, 2)$	$q(S^3) = 800K/3$
S_4 ={(200K,Los Angeles),(300K,Los Angeles),(300K,Los Angeles)}	$V_4 = (0, 1, 2)$	$q(S^4) = 800K/3$
S_5 ={(200K,Ann Arbor) ,(200K,Los Angeles),(300K,Los Angeles)}	$V_5 = (1, 1, 1)$	$q(S^5) = 900K/4$
S_6 ={(200K,Ann Arbor) ,(200K,Ann Arbor) ,(200K,Ann Arbor) }	$V_6 = (3, 0, 0)$	$q(S^6) = 200K$

Thus, using equation (6), the bias will be estimated as:

$$b \approx \frac{1}{6}\sum_{i=1}^{6} q(S_i) - q(S) = 251.4K - 225K = 26.4K$$

To use equation (7), we have $\bar{V} = (\frac{500K}{6}, \frac{100K}{2}, \frac{500K}{3})$, and:

$$q(\bar{V}) = 100K * \frac{2 + \frac{(1/2)*2+(5/3)*3}{(1/2)+(5/3)}}{2} = 238.5K$$

Now, we can use equation (7) to estimate the bias as:

$$b \approx \frac{1}{6}\sum_{i=1}^{6} q(S_i) - q(\bar{V}) = 251.4K - 238.5K = 12.9K$$

This estimate is closer to the real bias, which as computed in (3) is $b = 8.4K$.

In this example, although there is equal probability to bootstrap each of the tuples (200K, Ann Arbor), (200K, Los Angeles), and (300K, Los Angeles) from S, our bootstrap resamples included (300K, Los Angeles) more frequently than the other two tuples. Therefore, by recentering S to fit the empirical distribution formed by these 6 bootstrap resamples, the probability of bootstrapping (300K, Los Angeles) is enforced to be significantly higher than the probability of bootstrapping the other two items.

7 Communicating Error to Non-Statisticians

In general, for a SQL query with m aggregate columns in its `SELECT` clause, each output row is associated with $m + 1$ error terms: one error will capture the row's probability of existence, while others will capture the approximation quality of the aggregate columns in the row. Using confidence intervals as a common means of expressing aggregation error, we will need to (at least conceptually) append $2m + 1$ columns to the output relation of an approximate query, where each aggregation error will be captured using two columns: one for the confidence level α (e.g., 95%) and one for half of the interval width β. In other words, each output tuple T will have the following attributes:

$$T : B_1, \cdots, B_k, A_1, \cdots, A_m, \alpha_1, \cdots, \alpha_m, \beta_1, \cdots, \beta_m, p_e$$

where $B_1, \cdots, B_k$ are the non-aggregate columns in the output, $A_1, \cdots, A_m$ are the aggregate columns, α_i and β_i are the confidence level and half interval width for A_i, respectively. Finally, p_e is the probability of existence for tuple T (see Section 5). For example, assuming A_i is an unbiased estimate, with a probability of α, the true answer will be in the $[A_i - \beta_i, A_i + \beta_i]$ range.

While providing a rich set of accuracy measures, this scheme suffers from two major drawbacks:

1. Adding additional columns to the query output will break existing BI (Business Intelligence) tools designed to work with traditional database systems. Unless such tools are modified to expect (and ignore or use) the additional columns, they cannot work with an AQP system that modifies the output schema of the queries. In contrast, an AQP system that keeps the output schema intact will enjoy a greater degree of adoption: any off-the-shelf reporting and BI tool can be sped up by simply replacing its underlying RDBMS with such an AQP system, without the tool having to be aware of the fact that it is using an AQP database.

2. While a statistician might be able to understand, combine, and use these rich set of error statistics to perform sophisticated forms of inference, a typical database user will simply find a large number of errors associated with each row overwhelming. In fact, the majority of non-statisticians (e.g., developers and practitioners) may even find it difficult to interpret statistical error guarantees, such as confidence intervals and asymptotic errors.[12]

On the other hand, there is clear advantage to provide and use these error estimates when relying on approximate answers for decision making. While there is a clear need for more research in this area (i.e., how to best communicate statistical errors to a non-statisticians), here we outline a basic solution to this dilemma, which we call a *high-level accuracy contract (HAC)*.

High-level Accuracy Contract (HAC) — First, the AQP system calculates all these errors but, by default, does not add them to the query output. The query, however, can access these error estimates by explicitly invoking a designated function. For example,

```
SELECT call_center_id, AVG(review) AS satisfaction
FROM customer_service_logs
WHERE satisfaction < 2.0
      AND existence_probability() > 0.95
      AND relative_error(satisfaction, 0.95, 0.3)
GROUP BY call_center_id
ORDER BY satisfaction
);
```

where `existence_probability()` and `relative_error()` are special functions provided by the AQP engine. This will simply allow users (and BI tools) to decide if and when they would like to use the error estimates. However, to allow practitioners and existing BI tools that are not ready or willing to explicitly invoke such functions, the AQP system provides a high-level accuracy contract, expressed as a single number ϕ, where $0\phi \leq 1$. Once a high-level accuracy contract is set to a particular ϕ, the AQP system guarantees that any results returned to the users will be at least $\phi \times 100\%$ accurate. This is enforced as follows.

The AQP system simply omits every output tuple whose probability of existence is below ϕ. However, to deal with an aggregate value that does not meet this requested HAC[13] there are several policies that the AQP

[12] One solution is to substitute β_i with $\frac{\beta_i}{|A_i|} \times 100\%$ (when $A_i > 0$) for a relative error or with $\pm\beta_i$ for an absolute error. Even better, one might use a graphical interface where these numbers are replaced with error bars [32]. However, grasping the statistical significance of all these error bars will still remain overwhelming.

[13] An aggregate value does not meet the HAC if $\frac{\beta_i}{max\{|A_i|,\epsilon\}} < \phi$, where $\epsilon > 0$ is a small constant to prevent unduly small values of A_i from causing large relative errors. α_i can be also set to ϕ or to a default value, e.g., 0.95 or 0.99.

system can adopt:

- *Do nothing.* The AQP system simply returns all the aggregate values (possibility with a different display color if a graphical terminal is used).
- *Use a special symbol.* The system replaces the aggregate value with a special value, e.g., `NULL`, -1, or other user-specified values.
- *Drop the row.* The system drops the entire row if any of the aggregate columns in that row do not meet the required HAC.
- *Fail.* The system drops the entire output relation and throws an exception (e.g., through a SQL error) if any of the aggregate columns in any of the rows do not meet the required HAC.

The user can control the AQP system's behavior by choosing any of these policies, with 'do nothing' being the most lenient approach and 'fail' being the strictest behavior. In the latter, the user can decide whether to re-run the query with a more lenient policy or with a larger sample size, or simply resort to exact query evaluation. The downside of the 'drop the row' policy is that it will affect the internal logic of the BI tools if they keep track of the row count of the output relation. The 'use a special symbol' does not change the output cardinality, however, it will need a special symbol that is not present in the original data. The main advantages of the HAC approach is that (i) it allows AQP systems to be used as a drop-in solution for existing BI tools, (ii) enables practitioners and developers to express their required level of accuracy in an intuitive fashion, i.e., as a single percentage, (iii) provides a range of intuitive policies to cater to different levels of accuracy concerns, and (iv) allows advanced users to still access and use the detailed error statistics if they want to.

8 Conclusion

With the Big Data on the rise, there has been much demands in the recent years for building and using Approximate Query Processing (AQP) systems. Despite the rich history of database research in this area, there are many technical subtitles and open challenges that need to be properly addressed in order to build sound, efficient, and deployable AQP systems. In this chapter, for the first time, we focused on these subtle challenges, ranging from the marketing opportunities for AQP systems to their general anatomy and architecture. Considering that most database developers and practitioners are not statisticians, we provided a self-contained and self-explanatory cheatsheet for error estimation under different sampling strategies. At the same time, we explained a few major types of error in these systems—that tend to be easily overlooked—using simple examples. Finally, we introduced our *high-level accuracy contract (HAC)* approach to prevent the complexity of different types of error from overwhelming the end user of an AQP system.

Acknowledgements

The authors are grateful to Christopher Jermaine for encouraging them to write this article, to Jia Deng and Jacob Abernethy for their great feedback, to Suchee Shah for her helpful comments, and to many researchers in the database community who have helped advance the field of AQP systems through their contributions over the years. This work was in part supported by Amazon AWS, Microsoft Azure, and NSF (Grant No. CPS 1544844).

References

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

[2] Good data. http://www.gooddata.com/.

[3] IDC report. http://tinyurl.com/lvup9e4.

[4] Presto: Distributed SQL query engine for big data. https://prestodb.io/docs/current/release/release-0.61.html.

[5] SnappyData. http://www.snappydata.io/.

[6] Tableau software. http://www.tableausoftware.com/.

[7] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In *VLDB*, 1999.

[8] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *ACM SIGMOD*, 2000.

[9] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.

[10] 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 *SIGMOD*, 2014.

[11] 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*, 2013.

[12] 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*, 2012.

[13] M. Al-Kateb and B. S. Lee. Stratified Reservoir Sampling over Heterogeneous Data Streams. In *SSDBM*, 2010.

[14] S. Arya and D. M. Mount. Approximate range searching. In *SCG*, 1995.

[15] B. Babcock and S. Chaudhuri. Towards a robust query optimizer: A principled and practical approach. In *SIGMOD*, 2005.

[16] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *VLDB*, 2003.

[17] C. M. Bishop. *Pattern recognition and machine learning*. springer, 2006.

[18] V. Chandrasekaran and M. I. Jordan. Computational and statistical tradeoffs via convex relaxation. *CoRR*, abs/1211.1073, 2012.

[19] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *PODS*, 2000.

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

[21] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4, 2012.

[22] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55, 2005.

[23] N. Dalvi and D. Suciu. Efficient query evaluation on probabilistic databases. *The VLDB Journal*, 16(4), 2007.

[24] A. Dasgupta, P. Drineas, B. Harb, R. Kumar, and M. W. Mahoney. Sampling algorithms and coresets for \ell_p regression. *SIAM Journal on Computing*, 38, 2009.

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

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

[27] D. Fisher. Incremental, approximate database queries and uncertainty for exploratory visualization. In *LDAV*, 2011.

[28] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS*, 2008.

[29] R. Gemulla. Sampling algorithms for evolving datasets. 2008.

[30] S. Guha and B. Harb. Wavelet synopsis for data streams: minimizing non-euclidean error. In *KDD*, 2005.

[31] S. Har-Peled and S. Mazumdar. On coresets for k-means and k-median clustering. In *STOC*, 2004.

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

[33] D. G. Horvitz and D. J. Thompson. A generalization of sampling without replacement from a finite universe. *Journal of the American Statistical Association*, 47, 1952.

[34] Y. Hu, S. Sundara, and J. Srinivasan. Estimating Aggregates in Time-Constrained Approximate Queries in Oracle. In *EDBT*, 2009.

[35] S. Joshi and C. Jermaine. Robust Stratified Sampling Plans for Low Selectivity Queries. In *ICDE*, 2008.

[36] S. Joshi and C. Jermaine. Sampling-Based Estimators for Subset-Based Queries. *VLDB J.*, 18(1), 2009.

[37] A. Kim, E. Blais, A. G. Parameswaran, P. Indyk, S. Madden, and R. Rubinfeld. Rapid sampling for visualizations with ordering guarantees. *PVLDB*, 2015.

[38] R. Kohavi et al. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Ijcai*, volume 14, 1995.

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

[40] J. Laurikkala. *Improving identification of difficult small classes by balancing class distribution*. Springer, 2001.

[41] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. In *Computer Graphics Forum*, volume 32, 2013.

[42] S. Matusevych, A. Smola, and A. Ahmed. Hokusai-sketching streams in real time. *arXiv preprint arXiv:1210.4891*, 2012.

[43] X. Meng. Scalable simple random sampling and stratified sampling. In *ICML*, 2013.

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

[45] M. Mohri, A. Rostamizadeh, and A. Talwalkar. *Foundations of machine learning*. MIT press, 2012.

[46] B. Mozafari. Verdict: A system for stochastic query planning. In *CIDR, Biennial Conference on Innovative Data Systems*, 2015.

[47] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. CliffGuard: A principled framework for finding robust database designs. In *SIGMOD*, 2015.

[48] B. Mozafari, E. Z. Y. Goh, and D. Y. Yoon. Cliffguard: An extended report. Technical report, University of Michigan, Ann Arbor, 2015.

[49] B. Mozafari and N. Niu. A handbook for building an approximate query engine. Technical report, University of Michigan, Ann Arbor, 2015.

[50] B. Mozafari, P. Sarkar, M. J. Franklin, M. I. Jordan, and S. Madden. Scaling up crowd-sourcing to very large datasets: A case for active learning. *PVLDB*, 8, 2014.

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

[52] C. Olston, S. Chopra, and U. Srivastava. Generating example data for dataflow programs. In *SIGMOD*, 2009.

[53] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online aggregation for large mapreduce jobs. *PVLDB*, 4, 2011.

[54] Y. Park, M. Cafarella, and B. Mozafari. Neighbor-sensitive hashing. *PVLDB*, 2015.

[55] Y. Park, M. Cafarella, and B. Mozafari. Visualization-aware sampling for very large databases. *CoRR*, 2015.

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

[57] N. Potti and J. M. Patel. DAQ: a new paradigm for approximate query processing. *PVLDB*, 8, 2015.

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

[59] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *CIDR*, 2011.

[60] H. Thakkar, N. Laptev, H. Mousavi, B. Mozafari, V. Russo, and C. Zaniolo. SMM: A data stream management system for knowledge discovery. In *ICDE*, 2011.

[61] S. K. Thompson. *Sampling*. Wiley series in probability and statistics. J. Wiley, 2012.

[62] Y. Tillé. *Sampling algorithms*. Springer, 2011.

[63] S. Vrbsky, K. Smith, and J. Liu. An object-oriented semantic data model to support approximate query processing. In *Proceedings of IFIP TC2 Working Conference on Object-Oriented Database Semantics*, 1990.

[64] S. Wu, B. C. Ooi, and K.-L. Tan. Continuous Sampling for Online Aggregation over Multiple Queries. In *SIGMOD*, pages 651–662, 2010.

[65] K. Zeng, S. Agarwal, A. Dave, M. Armbrust, and I. Stoica. G-OLA: Generalized on-line aggregation for interactive analysis on big data. In *SIGMOD*, 2015.

[66] K. Zeng, S. Gao, J. Gu, B. Mozafari, and C. Zaniolo. Abs: a system for scalable approximate queries with accuracy guarantees. In *SIGMOD*, 2014.

[67] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.

Scalable Analytics Model Calibration with Online Aggregation

Florin Rusu Chengjie Qin Martin Torres
University of California Merced
{frusu, cqin3, mtorres58}@ucmerced.edu

Abstract

Model calibration is a major challenge faced by the plethora of statistical analytics packages that are increasingly used in Big Data applications. Identifying the optimal model parameters is a time-consuming process that has to be executed from scratch for every dataset/model combination even by experienced data scientists. We argue that the lack of support to quickly identify sub-optimal configurations is the principal cause. In this paper, we apply parallel online aggregation to identify sub-optimal configurations early in the processing by incrementally sampling the training dataset and estimating the objective function corresponding to each configuration. We design concurrent online aggregation estimators and define halting conditions to accurately and timely stop the execution. The end-result is online approximate gradient descent—a novel optimization method for scalable model calibration. We show how online approximate gradient descent can be represented as generic database aggregation and implement the resulting solution in GLADE—a state-of-the-art Big Data analytics system.

1 Introduction

Big Data analytics is a major topic in contemporary data management and machine learning research and practice. Many platforms, e.g., OptiML [4], GraphLab [48, 49, 29], SystemML [3], SimSQL [50], Google Brain [25], GLADE [40, 12] and libraries, e.g., MADlib [26], Bismarck [17], MLlib [15], Vowpal Wabbit [1], have been proposed to provide support for distributed/parallel statistical analytics.

Model calibration is a fundamental problem that has to be handled by any Big Data analytics system. Identifying the optimal model parameters is an interactive, human-in-the-loop process that requires many hours – if not days and months – even for experienced data scientists. From discussions with skilled data scientists and our own experience, we identified several reasons that make model calibration a difficult problem. The first reason is that the entire process has to be executed from scratch for every dataset/model combination. There is little to nothing that can be reused from past experience when a new model has to be trained on an existing dataset or even when the same model is applied to a new dataset. The second reason is the massive size of the parameter space—both in terms of cardinality and dimensionality. Moreover, the optimal parameter configuration is dependent on the position in the model space. And third, parameter configurations are evaluated iteratively—one at a time. This is problematic because the complete evaluation of a single configuration – even sub-optimal ones – can take prohibitively long.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Gradient descent optimization [6] is a fundamental method for model calibration due to its generality and simplicity. It can be applied virtually to any analytics model [17] – including support vector machines (SVM), logistic regression, low-rank matrix factorization, conditional random fields, and deep neural networks – for which the gradient or sub-gradient can be computed or estimated. All the statistical analytics platforms mentioned previously implement one version or another of gradient descent optimization. Although there is essentially a single parameter, i.e., the step size, that has to be set in a gradient descent method, its impact on model calibration is tremendous. Finding a good-enough step size can be a time-consuming task. More so, in the context of the massive datasets and highly-dimensional models encountered in Big Data applications.

The standard practice of applying gradient descent to model calibration, e.g., MADlib, Vowpal Wabbit, MLlib, Bismarck, illustrates the identified problems perfectly. For a new dataset/model combination, an arbitrary step size is chosen. Model training is executed for a fixed number of iterations. Since the objective function is computed only for the result model – due to the additional pass over the data it incurs – it is impossible to identify bad step sizes in a smaller number of iterations. The process is repeated with different step values, chosen based on previous iterations, until a good-enough step size is found. Certain systems, e.g., Google Brain, support the evaluation of multiple step sizes concurrently. This is done by executing independent jobs on a massive cluster, without any sort of sharing.

We consider the abstract problem of *distributed model calibration with iterative optimization methods*, e.g., gradient descent. We argue that the incapacity to evaluate multiple parameter configurations simultaneously and the lack of support to quickly identify sub-optimal configurations are the principal causes that make model calibration difficult. It is important to emphasize that these problems are not specific to a particular model, but rather they are inherent to the optimization method used in training. The target of our methods is to find optimal configurations for the tunable hyper-parameters of the optimization method, e.g., step size, which, in turn, facilitate the discovery of optimal values for the model parameters.

In this paper, we apply parallel online aggregation to identify sub-optimal configurations early in the processing by incrementally sampling the training dataset and estimating the objective function corresponding to each configuration. We design concurrent online aggregation estimators and define halting conditions to accurately and timely stop the execution. We provide efficient parallel solutions for the evaluation of the concurrent estimators and of their confidence bounds that guarantee fast convergence. The end-result is online approximate gradient descent—a novel optimization method for scalable model calibration. We show how online approximate gradient descent can be represented as generic database aggregation and implement the resulting solution in GLADE—a state-of-the-art Big Data analytics system. Our major contribution is the conceptual integration of parallel multi-query processing and online aggregation for efficient and effective large-scale model calibration with gradient descent methods. This requires novel technical solutions as well as engineering artifacts in order to bring significant improvements to the state-of-the-art.

The article is organized as follows. We begin with a brief overview of GLADE—a parallel data processing system targeted at aggregate computation (Section 2). GLADE provides a series of features that are essential for designing and implementing online approximate gradient descent. Section 3 provides an overview of parallel online aggregation and how it is supported in GLADE. Section 4 presents online approximate gradient descent—the novel optimization method for analytics model calibration. A concrete application to SVM classification is given in Section 5. We conclude with future directions in Section 5.

2 A Brief Overview of GLADE

GLADE is a parallel data processing system executing any computation specified as a Generalized Linear Aggregate (GLA) [40, 12] using a merge-oriented strategy. It provides an infrastructure abstraction for parallel processing that decouples the algorithm from the runtime execution. The algorithm has to be specified in terms of the clean GLA interface, while the runtime takes care of all the execution details including data management,

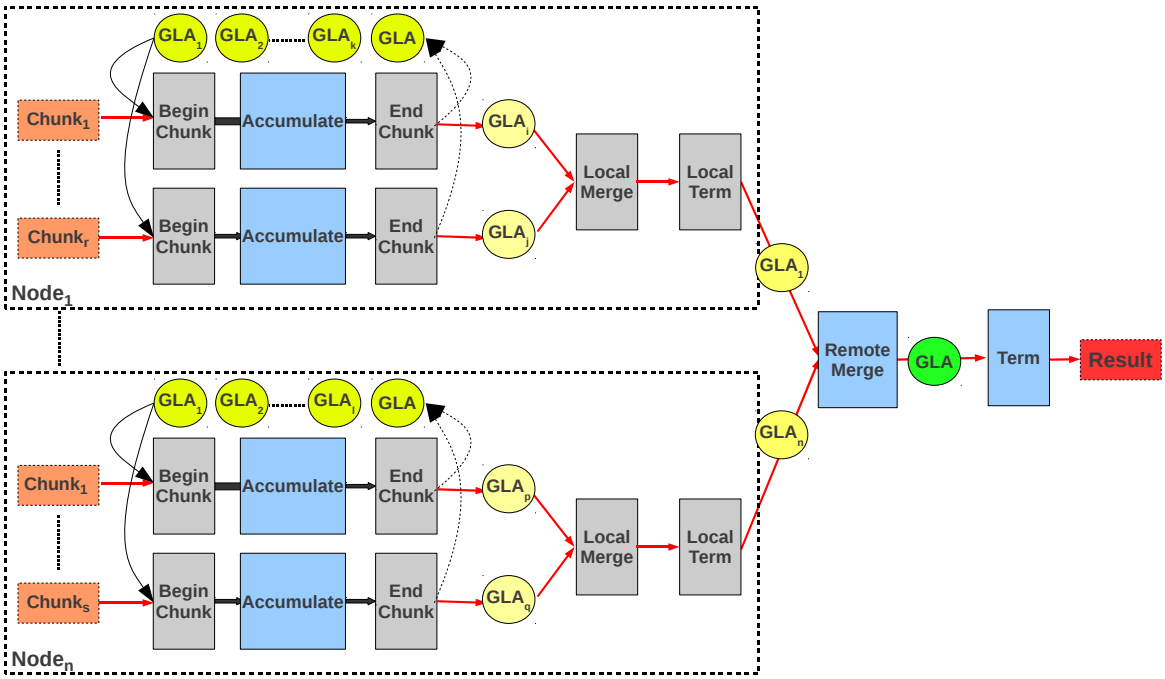


Figure 1: GLADE execution model implements the GLA abstraction.

memory management, and scheduling. GLADE is architecture-independent. It uses thread-level parallelism exclusively inside a processing node while process-level parallelism is used only across nodes. There is no difference between these two, though, in the GLADE infrastructure abstraction.

Execution Model Figure 1 depicts the GLADE execution model expressed in terms of the GLA interface abstraction [13]. GLA extend database User-Defined Aggregates (UDA) with methods for vectorized parallel processing. The standard UDA interface defines the following four methods: `Init`, `Accumulate`, `Merge`, and `Terminate`. These methods operate on the `state` of the object. While the interface is standard, the user has complete freedom when defining the `state` and implementing the methods.

GLADE evaluates GLAs by calling the interface methods as follows. `Init` – not displayed in Figure 1 – initializes the state of the object. `BeginChunk` is invoked before the data inside the chunk are processed—once for every chunk. `EndChunk` is similar to `BeginChunk`, invoked after processing the chunk instead. These two methods operate at chunk granularity. For example, data can be sorted for more efficient processing in `BeginChunk`, while temporary state can be discarded in `EndChunk`. `Accumulate` is invoked for every input tuple and updates the state according to the user-defined code. Merging is intended for use when the input is partitioned and multiple GLAs are computed—one for each partition. It takes as input two GLAs and merges their state into a combined GLA. Merging is invoked in two places. `LocalMerge` puts together local GLAs created on the same processing node, while `RemoteMerge` combines GLAs computed at different nodes. This distinction provides different optimization opportunities for shared-memory and shared-nothing parallel processing, respectively. `Terminate` is called after all the GLAs are merged together in order to finalize the computation, while `LocalTerminate` is invoked after the GLAs at a processing node are merged. `LocalTerminate` allows for optimizations when processing is confined to each node and no data transfer is required. It is important to notice that not all the interface methods have to be implemented for every GLA. We provide an illustrative GLA example in Section 5.

3 Parallel Online Aggregation

The idea in online aggregation is to compute only an estimate for an aggregate query based on a sample of the data [24]. In order to provide any useful information, though, the estimate is required to be accurate and statistically significant. Different from one-time estimation [18, 20] that may produce very inaccurate estimates for arbitrary queries, *online aggregation is an iterative process* in which a series of estimators with improving accuracy are generated. This is accomplished by including more data in estimation, i.e., increasing the sample size, from one iteration to another. The end-user can decide to run a subsequent iteration based on the accuracy of the estimator.

Method	**Usage**

`Init ()`
`Accumulate (Item` d`)`
`Merge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`
`Terminate ()`
	Basic UDA interface

`BeginChunk ()` `EndChunk ()`	Vectorized processing

`LocalMerge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`
`LocalTerminate ()`
`RemoteMerge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`
	Heterogeneous parallelism

`EstimatorTerminate ()` `EstimatorMerge (GLA` $input_1$`, GLA` $input_2$`, GLA` $output$`)`	Partial aggregate computation
`Estimate (`$estimator$`,` $lower$`,` $upper$`,` $confidence$`)`	Online estimation

Table 1: Estimation-enhanced GLA interface.

Overlap Query Processing and Estimation A more efficient alternative that avoids the sequential nature of the iterative process is to *overlap query processing with estimation* [16, 2]. As more data are processed towards computing the query result, the accuracy of the estimator improves accordingly. For this to be true, though, data are required to be processed in a statistically meaningful order, i.e., random order, to allow for the definition and analysis of the estimator. This is typically realized by randomizing data during the loading process. The apparent drawback of the overlapped approach is that the same query is essentially executed twice—once towards the final aggregate and once for computing the estimator. However, with the ever increasing number of cores available on modern CPUs, the additional computation power is not utilized unless concurrent tasks are found and executed. Given that the estimation process requires access to the same data as normal processing, estimation is a natural candidate for overlapped execution.

Design Space An online aggregation system provides estimates and confidence bounds during the entire query execution process. As more data are processed, the accuracy of the estimator increases while the confidence bounds shrink progressively, converging to the actual query result when the entire data have been processed. There are multiple aspects that have to be considered in the design of a parallel online aggregation system. First, a mechanism that allows for the computation of partial aggregates has to be devised. Second, a parallel sampling strategy to extract samples from data over which partial aggregates are computed has to be designed. Each sampling strategy leads to the definition of an estimator for the query result—estimator that has to be analyzed in order to derive confidence bounds. Each of these aspects is discussed in detail elsewhere [38]. In the

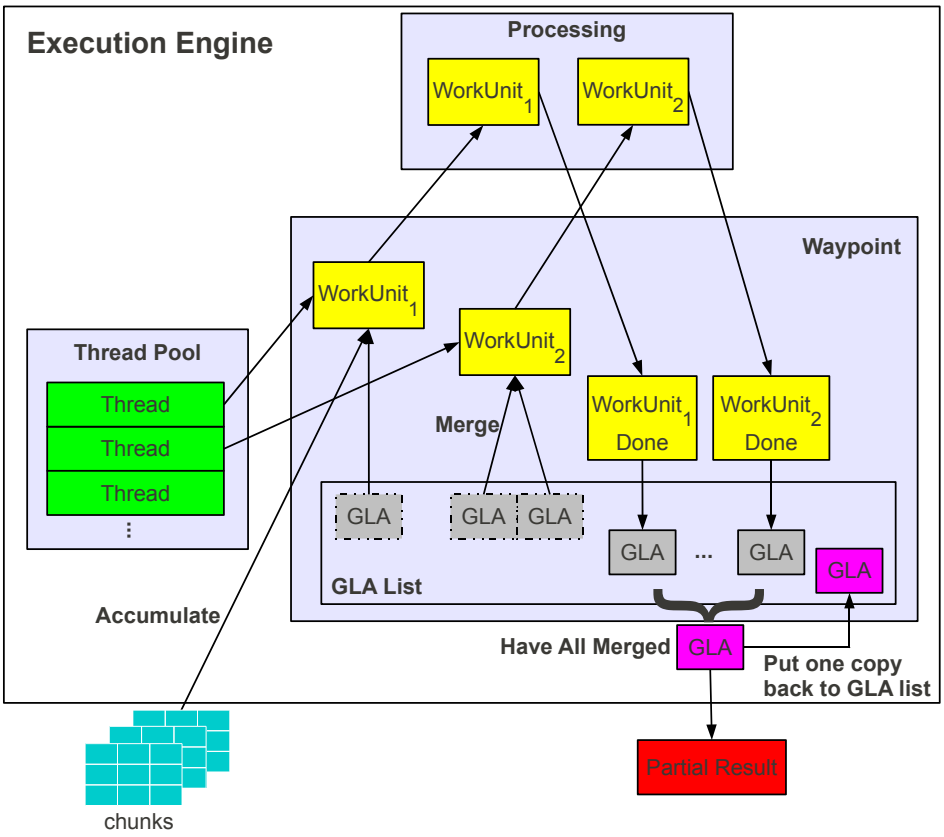


Figure 2: Execution strategy for parallel online aggregation in GLADE.

following, we present how online aggregation is supported in GLADE by enhancing the GLA abstraction with estimation capabilities.

3.1 Estimation-Enhanced GLA Abstraction

To support estimation, the GLA state has to be enriched with additional data on top of the original aggregate. Moreover, the GLA interface has to be also extended in order to distinguish between the final result and a partial result used for estimation. At least two methods have to be added—`EstimatorTerminate` and `EstimatorMerge`. `EstimatorTerminate` computes a local estimator at each node. It is invoked after merging the local GLAs during the estimation process. `EstimatorMerge` is called to put together in a single GLA the estimators computed at each node by `EstimatorTerminate`. It is invoked with GLAs originating at different nodes. Essentially, `EstimatorTerminate` and `EstimatorMerge` correspond to `LocalTerminate` and `RemoteMerge`, respectively. However, they contain exclusively estimation logic. The third method we add to the GLA interface is the `Estimate`. It is invoked by the user application on the GLA returned by the framework as a result of an estimation request. `Estimate` computes an estimator for the aggregate result and corresponding confidence bounds.

Table 1 summarizes the extended GLA interface we propose for parallel online aggregation. This interface abstracts both the aggregation and estimation processes in a reduced number of methods, releasing the user from the details of the actual execution in a parallel environment which are taken care of transparently by GLADE. Thus, the user can focus only on estimation modeling.

3.2 GLADE Implementation

Adding online aggregation to GLADE requires the extraction of a snapshot of the system state that can be used for estimation. Our solution overlaps the process of taking the snapshot with the actual GLA processing in order to have minimum impact on the overall execution time. This process consists of multiple stages. In the first stage, data are stored in random order on disk. This is an offline process executed at load time. Once data are available, we can start executing aggregate queries. The user application submits a query to the coordinator—a designated node managing the execution. It is the job of the coordinator to send the query further to the worker nodes, coordinate the execution, and return the result to the user application once the query is done. The result of a query is always a GLA containing the final state of the aggregate. When executing the query in non-interactive mode, the user application blocks until the final GLA arrives from the coordinator. When running in interactive mode with online aggregation enabled, the user application emits requests to the coordinator asking for the current state of the computation. A query always starts executing in non-interactive mode. Partial results are extracted asynchronously based exclusively on user application requests. The reason for this design choice is our goal to allow maximum asynchrony between the nodes in the system and to minimize the number of communication messages. It is clear that generating partial results interferes with the actual computation of the query result. However, in the case of aggregate computation, this is equivalent to early aggregation which is nonetheless executed as part of the final aggregation.

At a high level, enhancing GLADE with online aggregation is just a matter of providing support for the enhanced GLA interface in Table 1. The main challenge is how to overlap online estimation and actual query processing at all levels of the system. Abstractly, this corresponds to executing two simultaneous GLA computations. However, rather than treating actual computation and estimation as two separate GLAs, we group everything into a single enhanced GLA. This simplifies the control logic depicted in Figure 2. A partial result request triggers the merging of all existent GLAs. The resulting GLA is the partial result. Unlike the final result which is extracted from the waypoint and passed for further merging across the nodes, a copy of the partial result needs to be kept inside the waypoint and used for the final result computation. The newly arriving chunks are processed as before. The result is a completely new list of GLA states. The local GLA resulted through merging is added to this new list once the merging process ends.

4 Analytics Model Calibration

Consider the following model calibration problem with a linearly separable objective function:

$$\Lambda(\vec{w}) = min_{w \in \mathbb{R}^d} \sum_{i=1}^{N} f\left(\vec{w}, \vec{x_i}; y_i\right) + \mu R(\vec{w}) \tag{8}$$

in which a d-dimensional vector $\vec{w}$, $d \geq 1$, known as the *model*, has to be found such that the objective function is minimized. The training dataset consists of N (vector, label) pairs $\{(\vec{x}_1, y_1), \ldots, (\vec{x}_N, y_N)\}$ partitioned into M subsets. Each subset is assigned to a different processing node for execution. We focus on the case when N is extremely large and each of the M subsets are disk-resident. The constants $\vec{x_i}$ and y_i, $1 \leq i \leq N$, correspond to the feature vector of the i$^{\text{th}}$ data example and its scalar label, respectively. Function f is known as the loss while R is a regularization term to prevent overfitting. μ is a constant.

4.1 Gradient Descent Optimization

Gradient descent represents, by far, the most popular method to solve the class of optimization problems given in Eq. (8). Gradient descent is an iterative optimization algorithm that starts from an arbitrary point $\vec{w}^{(0)}$ and computes new points $\vec{w}^{(k+1)}$ such that the loss decreases at every step, i.e., $f(w^{(k+1)}) < f(w^{(k)})$. The new

points $\vec{w}^{(k+1)}$ are determined by moving along the opposite Λ gradient direction. Formally, the Λ gradient is a vector consisting of entries given by the partial derivative with respect to each dimension, i.e., $\nabla\Lambda(\vec{w}) = \left[\frac{\partial\Lambda(\vec{w})}{\partial w_1}, \ldots, \frac{\partial\Lambda(\vec{w})}{\partial w_d}\right]$. Computing the gradient for the formulation in Eq. (8) reduces to the gradient computation for the loss f and the regularizer R, respectively. The length of the move at a given iteration is known as the step size, denoted by $\alpha^{(k)}$. Gradient descent is highly sensitive to the choice of the *hyper-parameter* $\alpha^{(k)}$ which requires careful tuning for every dataset. With these, we can write the recursive equation characterizing any gradient descent method:

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha^{(k)}\nabla\Lambda\left(\vec{w}^{(k)}\right) \tag{9}$$

In order to check for convergence, the objective function Λ has to be evaluated at $\vec{w}^{(k+1)}$ after each iteration. Convergence to the global minimum is guaranteed only when Λ is convex. The process is depicted in Figure 3(a).

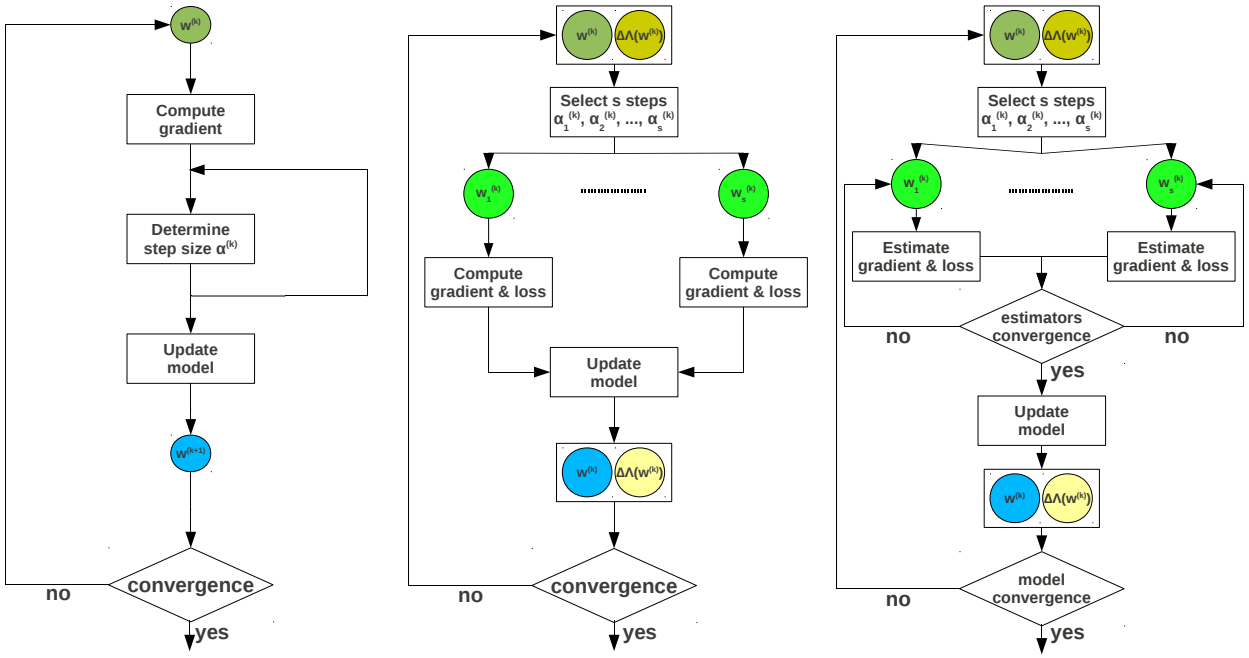
Batch Gradient Descent (BGD) The direct implementation of gradient descent is known as Batch Gradient Descent (BGD). We adopt it in this work due to its applicability to online aggregation. A detailed discussion of other alternatives is available in [11, 36]. The standard approach to compute the updated model $\vec{w}^{(k+1)}$ in BGD – once the direction of the gradient is determined – is to use line search methods which require objective and gradient loss evaluation. These involve multiple passes over the entire data. A widely used alternative is to fix the step size to some arbitrary value and then decrease it as more iterations are executed. The burden is essentially moved from runtime evaluation to offline tuning.

Parallel BGD The straightforward strategy to parallelize BGD is to overlap gradient computation across the processing nodes storing the M data subsets [7]. The partial gradients are subsequently aggregated at a coordinator node holding the current model $\vec{w}^{(k)}$, where a single global update step is performed in order to generate the new model $\vec{w}^{(k+1)}$ based on Eq. (9). The update step requires completion of partial gradient computation across all the nodes, i.e., update model is a synchronization barrier. Once the new model $\vec{w}^{(k+1)}$ is computed – using line search or a fixed step size – it is disseminated to the processing nodes for a new iteration over the data. Parallel BGD works because the objective function Λ is linearly separable, i.e., the gradient of a sum is the sum of the gradient applied to each term. It provides linear processing speedup and logarithmic communication in the number of nodes. In terms of convergence, though, there is no improvement with respect to the sequential algorithm—only faster iterations for gradient computation.

4.2 Speculative Gradient Descent

Speculative gradient descent addresses two fundamental problems—hyper-parameter tuning and convergence detection. Gradient descent depends on a series of hyper-parameters—the most important of which is the step size. Finding the optimal value for the hyper-parameters typically requires many trials. While convergence detection requires loss evaluation at every iteration, the standard practice, e.g., Vowpal Wabbit [1], MLLib [15], is to discard detection altogether and execute the algorithm for a fixed number of iterations. The reason is simple: loss computation requires a complete pass over the data. Discarding loss computation increases both the number of trials in hyper-parameter tuning as well as the duration of each trial.

The main idea in speculative gradient descent is to *overlap gradient and loss computation for multiple hyper-parameter configurations across every data traversal*. The number of hyper-parameter configurations used at each iteration is determined adaptively and dynamically at runtime. Their values are drawn from parametric distributions that are continuously updated using a Bayesian statistics approach. Only the model with the minimum loss survives each iteration, while the others are discarded. Speculative gradient descent allows for early bad configuration identification since many trials are executed simultaneously and timely convergence detection. Overall, faster model calibration.



(a) (b) (c)

Figure 3: Gradient descent optimization: (a) batch; (b) speculative; (c) online approximate.

Figure 3(b) depicts the speculative BGD algorithm. It starts with a set of possible step sizes $\{\alpha_1, \ldots, \alpha_s\}$, $s \geq 1$. An updated model $\vec{w}_i^{(k+1)}$, $1 \leq i \leq s$ is generated for each of these step sizes and the corresponding loss functions are computed concurrently. The model $\vec{w}_i^{(k+1)}$ with the minimum loss is chosen as the new model. The possible values of the step sizes are updated according to the computed loss function values. The procedure is repeated until convergence.

4.3 Online Approximate Gradient Descent

Speculative gradient descent allows for a more effective exploration of the parameter space. However, it still requires complete passes over the entire data at each iteration. *Online approximate gradient descent applies online aggregation sampling to avoid this whenever possible and identify the sub-optimal hyper-parameter configurations earlier.* This works because both the gradient and the loss can be expressed as aggregates in the case of linearly separable objective functions. The most important challenges we have to address are how to compute multiple concurrent sampling estimators corresponding to speculative gradient and loss computation (Figure 4) and how to design halting mechanisms that allow for the speculative query execution to be stopped as early as a user-defined accuracy threshold is achieved. A discussion on concurrent estimators for gradient and loss is available in [11]. In the following, we present the halting mechanisms.

When to stop gradient computation? In the case of gradient computation, there are d aggregates—one for every dimension in the feature vector. With online aggregation active, there is an estimator and corresponding confidence bounds for every aggregate. Given a desired level of accuracy, we have to determine when to stop the computation and move to a new iteration. We measure the accuracy of an estimator by the *relative error*, defined as the ratio between the confidence bounds width and the estimate, i.e., $\frac{high-low}{estimate}$. For example, if we aim for

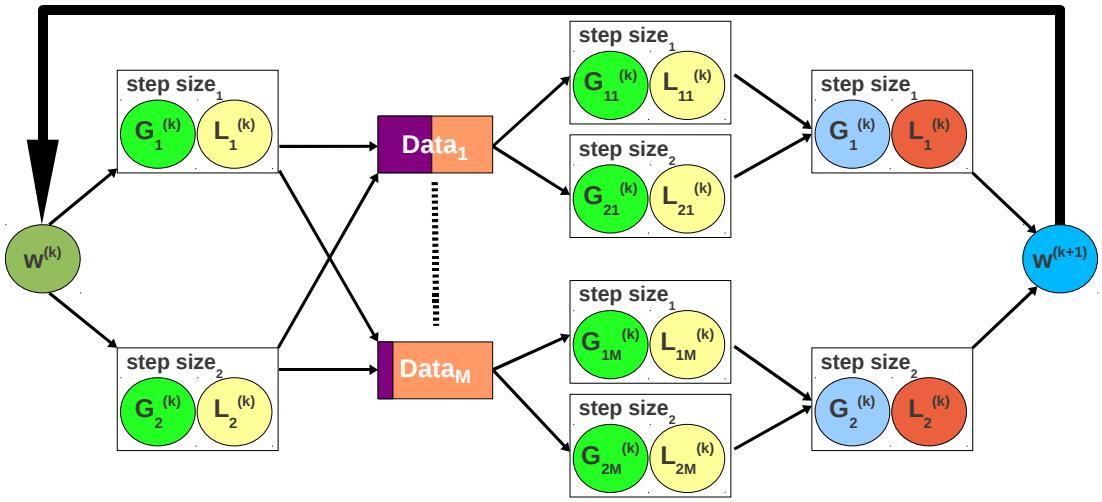


Figure 4: Parallel GLA evaluation for online approximate gradient descent.

95% accuracy, the relative error has to be below the user-defined threshold $\epsilon = 0.05$. What makes our problem difficult is that we have d independent relative errors, one for each estimator. The simple solution is to wait until all the errors drop below the threshold ϵ. This might require processing the entire dataset even when only a few estimators do not satisfy the accuracy requirement, thus defeating the purpose of online aggregation. An alternative that eliminates this problem is to require that only a percentage of the estimators, e.g., 90%, achieve the desired accuracy. Another alternative is to define a single convergence threshold across the d estimators. For example, we can define $\epsilon' = d \cdot \epsilon$ and require that the sum of the relative errors across estimators is below ϵ'.

When to stop loss computation? While the estimators in gradient computation are independent – in the sense that there is no interaction between their confidence bounds with respect to the stopping criterion – the loss estimators corresponding to different step sizes are dependent. Our goal is to choose only the estimator generating the minimum loss. Whenever we determine this estimator with high accuracy, we can stop the execution and start a new iteration. Notice that finding the actual loss – or an accurate approximation of it – is not required if gradient descent is executed for a fixed number of iterations. We design the following algorithm for stopping loss computation. The idea is to prune as many estimators as possible early in the execution. It is important to emphasize that pruning impacts only the convergence rate—not the correctness of the proposed method. Pruning conditions are exact and approximate. All the estimators for which there exists an estimator with confidence bounds completely below their confidence bounds, can be pruned. The remaining estimators overlap. In this situation, we resort to approximate pruning. We consider three cases. First, if the overlap between the upper bound of one estimator and the lower bound of another is below a user-specified threshold, the upper estimator can be discarded with high accuracy. Second, if an estimator is contained inside another at the upper-end, the contained estimator can be discarded. The third case is symmetric, with the inner estimator contained at the lower-end. The encompassing estimator can be discarded in this case. The algorithm is executed every time a new series of estimators are generated (Figure 3(c)). Execution can be stopped when a single estimator survives the pruning process. If the process is executed until the estimators achieve the desired accuracy, we choose the estimator with the lowest expected value.

When to stop gradient & loss computation? Remember that gradient and loss computation are overlapped in speculative gradient descent. When we move from a given point, we compute both the loss and the gradient at all the step sizes considered. In the online aggregation solution, we compute estimators and confidence bounds

for each of these quantities. The goal is to stop the overall computation as early as possible. How do we achieve this? We have to combine the stopping criteria for gradient and loss computation. The driving factor is loss computation. Whenever a step size can be discarded based on the exact pruning condition, the corresponding gradient estimation can be also discarded. Instead of applying the approximate pruning conditions directly, we have to consider the interaction with gradient estimation. While gradient estimation for the minimum loss does not converge, we continue the estimation for all the step sizes that cannot be discarded. This allows us to identify the minimum loss and its corresponding gradient with higher accuracy.

Algorithm 1 *GLA Online Approximate Gradient Descent*

State:

starting model: $\vec{w}$
s candidate models: $\vec{w_1}, \vec{w_2}, \ldots, \vec{w_s}$
s step sizes: $\alpha_1, \alpha_2, \ldots, \alpha_s$
s gradients: $\nabla\Lambda_1, \nabla\Lambda_2, \ldots, \nabla\Lambda_s$
s loss values: $\Lambda_1, \Lambda_2, \ldots, \Lambda_s$
estimation statistics: *gradient_stats*, *loss_stats*

Init ()

1. Initialize *gradient_stats*, *loss_stats*
2. Reset loss values $\Lambda_1, \Lambda_2, \ldots, \Lambda_s$
3. Calculate candidate models $\vec{w_1}, \vec{w_2}, \ldots, \vec{w_s}$ from $\vec{w}$ based on step sizes $\alpha_1, \alpha_2, \ldots, \alpha_s$

Accumulate (Item d)

1. Aggregate d to $\nabla\Lambda_1, \nabla\Lambda_2, \ldots, \nabla\Lambda_s$
2. Aggregate d to $\Lambda_1, \Lambda_2, \ldots, \Lambda_s$
3. Update *gradient_stats*, *loss_stats* with d

Merge (GLA $input_1$, GLA $input_2$, GLA *output*)

1. Merge $input_1.\{\nabla\Lambda_1, \ldots, \nabla\Lambda_s\}$ and $input_2.\{\nabla\Lambda_1, \ldots, \nabla\Lambda_s\}$ into $output.\{\nabla\Lambda_1, \ldots, \nabla\Lambda_s\}$
2. Merge $input_1.\{\Lambda_1, \ldots, \Lambda_s\}$ and $input_2.\{\Lambda_1, \ldots, \Lambda_s\}$ into $output.\{\Lambda_1, \ldots, \Lambda_s\}$

Terminate ()

1. Update $\vec{w}$ with the candidate model selected in *EstimatorTerminate*

EstimatorMerge (GLA $input_1$, GLA $input_2$, GLA *output*)

1. Merge $input_1.\{gradient\_stats\}$ and $input_2.\{gradient\_stats\}$ into $output.\{gradient\_stats\}$
2. Merge $input_1.\{loss\_stats\}$ and $input_2.\{loss\_stats\}$ into $output.\{loss\_stats\}$

EstimatorTerminate ()

1. Select the candidate model $\vec{w_i}$ having the minimum estimated loss according to *loss_stats*

Estimate ($estimator$, $lower$, $upper$, $confidence$)

1. Compute estimate and confidence bounds for gradients $\nabla\Lambda_1, \ldots, \nabla\Lambda_s$ from *gradient_stats*
2. Compute estimate and confidence bounds for loss values $\Lambda_1, \ldots, \Lambda_s$ from *loss_stats*
3. Prune out candidate models with sub-optimal loss estimates
4. Stop estimation for converged candidate models

5 An Example: SVM Classification GLA

We give an illustrative example that shows how SVM classification – a standard analytics model – can be expressed as `SQL` aggregates. The online approximate gradient descent optimization method is applied for

training the model. We provide the corresponding GLA implemented in GLADE and experimental results that confirm the benefits of online aggregation for model calibration.

The objective function in SVM classification with -1/+1 labels and L_1-norm regularization is given by: $\sum_i \left(1 - y_i \vec{w}^T \cdot \vec{x_i}\right)_+ + \mu||\vec{w}||_1$. This can be written as a standard `SQL` aggregate:

```
SELECT SUM(∑_{i=1}^{d} 1 − yx_i w_i) FROM T
WHERE (∑_{i=1}^{d} 1 − yx_i w_i) > 0
```

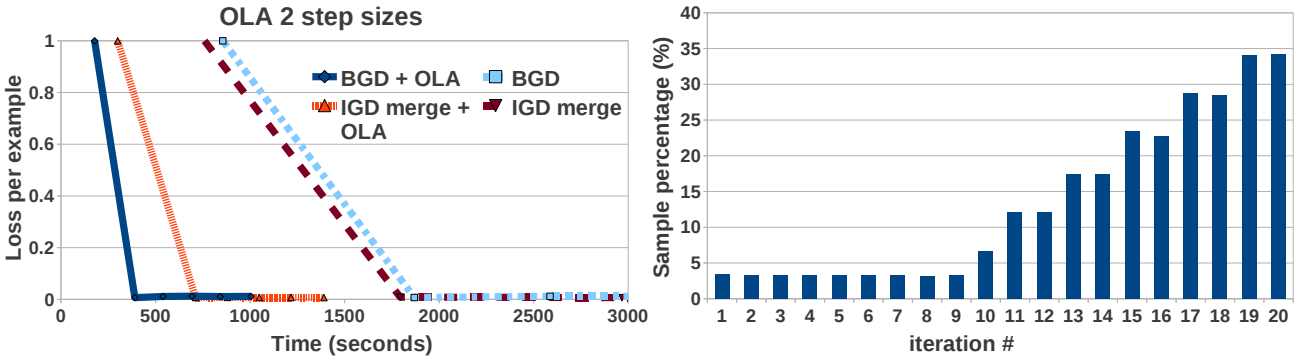
where `T` is a table that stores the training examples and their label. It is important to emphasize that model $\vec{w}$ is constant at any loss function evaluation. Gradient computation at point $\vec{w}$ requires one aggregate for every dimension, as shown in the following `SQL` query:

```
SELECT SUM(−yx_1), ..., SUM(−yx_d) FROM T
WHERE (∑_{i=1}^{d} 1 − yx_i w_i) > 0
```

In general, the BGD computations can be expressed as the following abstract `SQL` aggregate query:

```
SELECT SUM(f_1(t)), ..., SUM(f_p(t)) FROM T
```

in which p different aggregates are computed over the training tuples in relation T. This allows us to map BGD into a GLA and execute it in GLADE. The GLA corresponding to online approximate gradient descent is given in Algorithm 1.



(a) (b)

Figure 5: Experimental results for SVM classification: (a) convergence, (b) sampling ratio.

Figure 5 depicts the results of executing the online approximate gradient descent GLA for SVM classification over a 9-node GLADE cluster. The training dataset in these experiments is `splice` [11]. It consists of 50 million examples over a 13 million feature space. In Figure 5(a) we can observe the improvements in convergence rate online approximate gradient descent provides over standard BGD. Similar benefits translate to stochastic or incremental gradient descent (IGD). Figure 5(b) depicts the sampling ratio used at the time a decision is made about which candidate model is optimal. Far from the optimum, less than 5% sample is enough to identify the optimal candidate model. The ratio increases gradually as we get closer to the optimum. These results prove the benefits online aggregation can provide to analytics model training.

6 Related Work

In this paper, we apply online aggregation estimators to complex analytics, rather than focusing on standard SQL aggregates—the case in our previous work [38, 34]. We are the first to model gradient descent optimization as an aggregation problem. This allows us to design multiple concurrent estimators and to define halting mechanisms that stop the execution when model update and loss computation are overlapped. Moreover, the integration of online aggregation with speculative step evaluation allows for early identification of sub-optimal step sizes and directs the system resources toward the promising configurations. None of the existing systems support concurrent hyper-parameter evaluation or concurrent estimators. Our previous work on gradient descent optimization in GLADE [35, 37] is limited to stochastic gradient descent.

Online Aggregation The database online aggregation literature has its origins in the seminal paper by Hellerstein et al. [24]. We can broadly categorize this body of work into system design [39, 9, 16, 2], online join algorithms [23, 10, 43], online algorithms for estimations other than join [8, 47], and methods to derive confidence bounds [22]. All of this work is targeted at single-node centralized environments. The parallel online aggregation literature is not as rich though. We identified only three lines of research that are closely related to this paper. Luo et al. [19] extend the centralized ripple join algorithm [23] to a parallel setting. A stratified sampling estimator [14] is defined to compute the result estimate while confidence bounds cannot always be derived. Wu et al. [45] extend online aggregation to distributed P2P networks. They introduce a synchronized sampling estimator over partitioned data that requires data movement from storage nodes to processing nodes. In subsequent work, Wu et al. [44] tackle online aggregation over multiple queries. The third piece of relevant work is online aggregation in MapReduce. In [46], standard Hadoop is extended with a mechanism to compute partial aggregates. In subsequent work [31], an estimation framework based on Bayesian statistics is proposed. BlinkDB [42, 41] implements a multi-stage approximation mechanism based on pre-computed sampling synopses of multiple sizes, while EARL [30] and ABS [27] use bootstrapping to produce multiple estimators from the same sample.

Gradient Descent Optimization There is a plethora of work on distributed gradient descent algorithms published in machine learning [33, 25, 52, 21]. All these algorithms are similar in that a certain amount of model updates are performed at each node, followed by the transfer of the partial models across nodes. The differences lie in how the model communication is done [33, 52] and how the model is partitioned for specific tasks, e.g., neural networks [25] and matrix factorization [21]. Many of the distributed solutions are immediately applicable to multi-core shared memory environments. The work of Ré et al. [32, 17, 51] is representative in this sense. Our work is different from all these approaches because we consider concurrent evaluation of multiple step sizes and we use adaptive intra-iteration approximation to detect convergence. Moreover, stochastic gradient descent is taken by default to be the optimal gradient descent method, while BGD is hardly ever considered. Kumar et al. [28] have recently observed the benefits of BGD in training linear models over normalized relations that require join.

7 Conclusions and Future Work

In this paper, we apply parallel online aggregation to identify sub-optimal configurations early in the processing by incrementally sampling the training dataset and estimating the objective function corresponding to each configuration. We design concurrent online aggregation estimators and define halting conditions to accurately and timely stop the execution. The end-result is online approximate gradient descent—a novel optimization method for scalable model calibration. In future work, we plan to extend the proposed techniques to other model calibration methods beyond gradient descent.

Many Big Data analytics systems and frameworks implement gradient descent optimization. Most of them target distributed applications on top of the Hadoop MapReduce framework, e.g., Mahout [5], MLlib [15], while others provide complete stacks, e.g., MADlib [26], Distributed GraphLab [49], and Vowpal Wabbit [1]. With no exception, stochastic gradient descent is the primary method implemented in all these systems. As a first step, the techniques we present in this paper can be incorporated into any of these systems, as long as multi-threading parallelism and partial aggregation are supported. We plan to explore how this can be done in future work. More important, we provide strong evidence that BGD deserves full consideration in any Big Data analytics system.

Acknowledgments The work in this paper was supported in part by a Hellman Faculty Fellowship, a gift from LogicBlox, and a US Department of Energy (DOE) Early Career Award.

References

[1] A. Agarwal et al. A Reliable Effective Terascale Linear Learning System. *JMLR*, 15(1), 2014.

[2] A. Dobra et al. Turbo-Charging Estimate Convergence in DBO. *PVLDB*, 2(1), 2009.

[3] A. Ghoting et al. SystemML: Declarative Machine Learning on MapReduce. In *ICDE 2011*.

[4] A. Sujeeth et al. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML 2011*.

[5] Apache Mahout. `https://mahout.apache.org`. [Online; accessed July 2014].

[6] D. P. Bertsekas. Incremental Gradient, Subgradient, and Proximal Methods for Convex Optimization: A Survey. MIT 2010.

[7] C. Chu, S. Kim, Y. Lin, Y. Yu, G. Bradski, A. Ng, K. Olukotun. Map-Reduce for Machine Learning on Multicore. In *NIPS 2007*.

[8] C. Jermaine et al. Online Estimation for Subset-Based SQL Queries. In *VLDB 2005*.

[9] C. Jermaine et al. Scalable Approximate Query Processing with the DBO Engine. In *SIGMOD 2007*.

[10] C. Jermaine et al. The Sort-Merge-Shrink Join. *TODS*, 31(4), 2006.

[11] C. Qin and F. Rusu. Speculative Approximations for Terascale Analytics. `http://arxiv.org/abs/1501.00255`, 2015.

[12] Y. Cheng, C. Qin, and F. Rusu. GLADE: Big Data Analytics Made Easy. In *SIGMOD 2012*.

[13] Y. Cheng and F. Rusu. Formal Representation of the SS-DB Benchmark and Experimental Evaluation in EXTASCID. *DAPD*, 2014.

[14] W. G. Cochran. *Sampling Techniques*. Wiley, 1977.

[15] E. Sparks et al. MLI: An API for Distributed Machine Learning. In *ICDM 2013*.

[16] F. Rusu et al. The DBO Database System. In *SIGMOD 2008*.

[17] X. Feng, A. Kumar, B. Recht, and C. Ré. Towards a Unified Architecture for in-RDBMS Analytics. In *SIGMOD 2012*.

[18] G. Cormode et al. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4, 2012.

[19] G. Luo, C. J. Ellmann, P. J. Haas, and J. F. Naughton. A Scalable Hash Ripple Join Algorithm. In *SIGMOD 2002*.

[20] M. N. Garofalakis and P. B. Gibbon. Approximate Query Processing: Taming the TeraBytes. In *VLDB 2001*.

[21] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-Scale Matrix Factorization with Distributed Stochastic Gradient Descent. In *KDD 2011*.

[22] P. J. Haas. Large-Sample and Deterministic Confidence Intervals for Online Aggregation. In *SSDBM 1997*.

[23] P. J. Haas and J. M. Hellerstein. Ripple Joins for Online Aggregation. In *SIGMOD 1999*.

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

[25] J. Dean et al. Large Scale Distributed Deep Networks. In *NIPS 2012*.

[26] J. Hellerstein et al. The MADlib Analytics Library: Or MAD Skills, the SQL. *PVLDB*, 2012.

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

[28] A. Kumar, J. F. Naughton, and J. M. Patel. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD 2015*.

[29] A. Kyrola, G. Blelloch, and C. Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI 2012*.

[30] N. Laptev et al. Early Accurate Results for Advanced Analytics on MapReduce. *PVLDB*, 5(10), 2012.

[31] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11), 2011.

[32] F. Niu, B. Recht, C. Ré, and S. J. Wright. A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS 2011*.

[33] O. Dekel et al. Optimal Distributed Online Prediction Using Mini-Batches. *JMLR*, 13(1), 2012.

[34] C. Qin and F. Rusu. Parallel Online Aggregation in Action. In *SSDBM 2013*.

[35] C. Qin and F. Rusu. Scalable I/O-Bound Parallel Incremental Gradient Descent for Big Data Analytics in GLADE. In *DanaC 2013*.

[36] C. Qin and F. Rusu. Speculative Approximations for Terascale Distributed Gradient Descent Optimization. In *DanaC 2015*.

[37] C. Qin and F. Rusu. Speeding-Up Distributed Low-Rank Matrix Factorization. In *CloudCom-Asia 2013*.

[38] C. Qin and F. Rusu. PF-OLA: A High-Performance Framework for Parallel Online Aggregation. *DAPD*, 32(3), 2014.

[39] R. Avnur et al. CONTROL: Continuous Output and Navigation Technology with Refinement On-Line. In *SIGMOD 1998*.

[40] F. Rusu and A. Dobra. GLADE: A Scalable Framework for Efficient Analytics. *OS Review*, 46(1), 2012.

[41] S. Agarwal et al. Knowing When You're Wrong: Building Fast and Reliable Approximate Query Processing Systems. In *SIGMOD 2014*.

[42] S. Agarwal et al. Blink and It's Done: Interactive Queries on Very Large Data. *PVLDB*, 5(12), 2012.

[43] S. Chen et al. PR-Join: A Non-Blocking Join Achieving Higher Early Result Rate with Statistical Guarantees. In *SIGMOD 2010*.

[44] S. Wu et al. Continuous Sampling for Online Aggregation over Multiple Queries. In *SIGMOD 2010*.

[45] S. Wu et al. Distributed Online Aggregation. *PVLDB*, 2(1), 2009.

[46] T. Condie et al. MapReduce Online. In *NSDI 2010*.

[47] M. Wu and C. Jermaine. A Bayesian Method for Guessing the Extreme Values in a Data Set. In *VLDB 2007*.

[48] Y. Low et al. GraphLab: A New Parallel Framework for Machine Learning. In *UAI 2010*.

[49] Y. Low et al. Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud. *PVLDB*, 5(8), 2012.

[50] Z. Cai et al. Simulation of Database-Valued Markov Chains using SimSQL. In *SIGMOD 2013*.

[51] C. Zhang and C. Ré. DimmWitted: A Study of Main-Memory Statistical Analytics. *PVLDB*, 7(12), 2014.

[52] M. Zinkevich, M. Weimer, A. Smola, and L. Li. Parallelized Stochastic Gradient Descent. In *NIPS 2010*.

The Complexity of Evaluating Order Queries with the Crowd*

Benoît Groz
Univ Paris-Sud
groz@lri.fr

Tova Milo
Tel Aviv University
milo@cs.tau.ac.il

Sudeepa Roy†
Duke University
sudeepa@cs.duke.edu

1 Introduction

One of the foremost challenges for information technology over the last few years has been to explore, understand, and extract useful information from large amounts of data. Some particular tasks such as annotating data or matching entities have been outsourced to human workers for many years. But the last few years have seen the rise of a new research field called *crowdsourcing* that aims at delegating a wide range of tasks to human workers, building formal frameworks, and improving the efficiency of these processes.

The database community has thus been suggesting algorithms to process traditional data manipulation operators with the crowd, such as joins or filtering. This is even more useful when comparing the underlying "tuples" is a subjective decision – *e.g.*, when they are photos, text, or simply noisy data with different variations and interpretations – and can presumably be done better and faster by humans than by machines.

The problems considered in this article aim to retrieve a subset of preferred items from a set of items by delegating pairwise comparison operations to the crowd. The most obvious example is finding the maximum of a set of items (called *max*). We also consider two natural generalizations of the max problem:

1. *Top-k:* computing the top-k items according to a given criterion (max = top-1), and
2. *Skyline:* computing all Pareto-optimal items when items can be ordered according to multiple criteria (max has a single criterion).

We assume that there is an underlying ground truth for all these problems, *i.e.*, the items have fixed values along all criteria that lead to well-defined solutions for these problems. However, these values are unknown and the only way to reach a solution is by asking the crowd to compare pairs of items. Toward this goal, we adopt a widespread and simple model for the crowd and present the prevalent theoretical ideas from the existing work in the literature that allow us to compute the preferred items in this setting. This simple model builds the foundation for a formal framework, allows us to do a rigorous analysis, and serves as the basis for understanding the more general and practical cases. We also outline the limitations of the current settings and algorithms, and identify some interesting research opportunities in this domain.

To illustrate the above problems, imagine some Mr. Smith has been provided with a voucher for a stay in a ski resort location of his choice from a set of locations. Mr Smith may wish to choose the location with the

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

*This work has been partially funded by the European Research Council under the FP7, ERC grant MoDaS, agreement 291071, the Israel Ministry of Science, and by the NSF award IIS-0911036.

†This work has been done while the author was at the University of Washington.

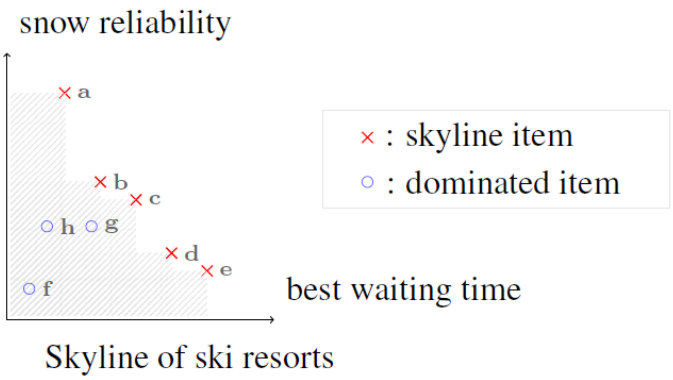


Figure 1: Example: Skyline of ski resorts

best possible crowd management (*i.e.*, where the average waiting time in the queue is the shortest). One possible way to reach a decision would be to ask people on the web about how the ski resorts compare to one another according to this criterion (pairwise comparisons). But to consider more options, he may also ask for the top-5 locations with respect to waiting time. And finally, if he cares about multiple criteria, like both the *waiting time* and the *snow reliability* of the ski resort at a given time, he may wish to remove from the list all locations that are beaten by another simultaneously on waiting time and snow reliability, and retain every location surviving this process for further examination (*e.g.*, to consider additional criteria like slopes, availability of cable cars and elevators, cleanliness, etc.). Figure 1 shows eight such fictive locations on these two criteria assuming that higher values are better for both criteria.

To achieve such goals, platforms like Amazon Mechanical Turk or CrowdFlower [40, 12, 39, 30, 11] help their customers connect to the crowd from all over the world through the Internet. The customers post *jobs* on these platforms, and the *workers* from the crowd solve multiple *tasks* for these jobs, and get paid for each task in exchange. For a given job (*e.g.*, sorting a list of items), the workers are given some tasks in rounds (*e.g.*, compare two given items). Once the answers to the tasks in the previous round are returned, some internal computation is done by the machine, the next set of tasks is decided, and given to the user in the next round. As the crowd is a more expensive resource compared to a machine, the *cost* of an algorithm in crowdsourcing setting is typically measured by the total cost paid to the workers while solving a job.

Workers gathered from the web, however, are prone to return erroneous answers. One major issue in crowdsourcing is therefore to mitigate those errors, typically through redundant tasks. In the absence of identified experts, the traditional "wisdom of the crowd" assumption in crowdsourcing means that the results aggregated from multiple workers will eventually converge toward the truth. Therefore, crowdsourcing algorithms endeavor to submit as few redundant tasks as possible to achieve a desired *accuracy* level, which bounds the probability of returning an incorrect result for the intended job. Sometimes, the *latency* of a crowdsourcing algorithm is also evaluated as a criterion in addition to cost and accuracy, which may include the number of rounds of tasks given to the workers, and also the workload in those rounds.

Obtaining precise cost, accuracy, and latency models for crowd workers by accurately estimating human behavior is a non-trivial task by itself. Different workers may have different levels of expertise and sincerity to solve a task, and may take different amounts of time in the process. Tasks can be assigned individually, or multiple tasks can be assigned together in a batch. The answers to various tasks returned by the same or different workers may be correlated. The customers may want to pay the same or different amounts of money for different tasks. In this article, however, we consider a simple and standard probabilistic setting to model the possible incorrect answers returned by the crowd, where the crowd is perceived as a *noisy comparison oracle*.

Crowd as a noisy comparison oracle. We assume that the items admit a correct, albeit implicit, solution

to each of the problems considered in this article (max, top-k, or skyline). We assume that values of the items (along single or multiple criteria) are not known explicitly and can only be recovered by asking the crowd to compare two items. In the simple and standard error model (called the *constant error model*), the crowd is considered as an oracle that always returns the correct answer with a constant probability $> \frac{1}{2}$ (say, $\frac{2}{3}$), *i.e.*, the oracle always performs better than a random answer. In other words, given two items x, y such that $x > y$, each *oracle call* $\mathcal{O}(x > y)$ will return $true$ with probability $\frac{2}{3}$ and $false$ with probability $\frac{1}{3}$. Further, the answers to two oracle calls are independent.

In this model, one can increase the confidence arbitrarily by repeating a query to the oracle and adopting majority vote. In practice, this means asking several people the same question. Every problem could be thus solved by repeating each oracle call a large number of times and adopting the majority vote. The challenge in this setting is to obtain more efficient algorithms by focusing on the most crucial questions to reduce redundancy on others. To control the error introduced by potentially incorrect oracle answers, all of our algorithms take as input some parameter $\delta < \frac{1}{2}$ called the *tolerance*. The algorithms will be designed with sufficient redundancy so that their final output is correct with probability at least $1 - \delta$. In the simple cost model, we will assume unit (*i.e.*, constant) cost per oracle call, and aim to optimize the total *number* of oracle calls; we will also discuss other interesting error and cost models in the following sections.

Roadmap. In Section 2, we discuss the max and top-k problems for noisy comparison oracles based on the work by Feige et al. [16] and Davidson et al. [13, 14]. In Section 3, we discuss the skyline problem for noisy comparisons based on the work by Groz and Milo [20]. We review the related work in Section 4 and conclude with future directions in Section 5.

2 Max and Top-k

First, we discuss the max and top-k problems in the setting with a noisy comparison oracle:

> **Input:** A set of items $S = \{x_1, \cdots, x_n\}$ in arbitrary order, where $x_1 > x_2 > \cdots > x_n$; tolerance δ; noisy comparison oracle $\mathcal{O}$; and $k \in [2, n]$ for top-k.
> **Objective:** (i) (**Max**) Return the maximum item (x_1) with error probability $\leq \delta$. (ii) (**Top-k**) Return $x_1, \cdots, x_k$ in order with error probability $\leq \delta$.

In Figure 1, the location that maximizes the experience with waiting time is location e, and the top-k locations according to this ordering for $k = 5$ are e, d, c, b, g.

2.1 Constant and Variable Error Model, Unit Cost Function

For the unit cost function, each call to the oracle for comparing two items has the same constant cost. Therefore, it suffices to count the number of calls to the oracle. Further, in practice, the comparisons can be delegated to one or more crowd workers arbitrarily without affecting the total cost. Next, we discuss two error models for the unit cost function.

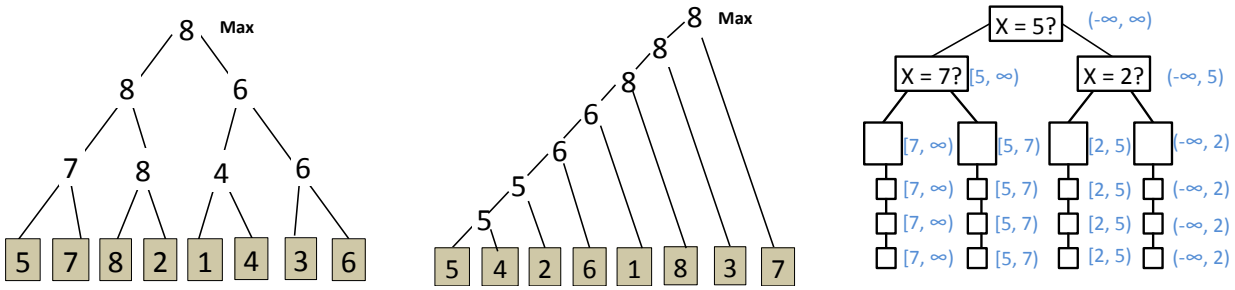
Constant Error Model

The *constant error model* is a standard error model for noisy comparison operations [16, 31, 19], where the probability of getting the correct answer from a noisy oracle is a constant $> \frac{1}{2}$, *i.e.*,

$$(\textbf{Constant error model}) \qquad \Pr[\mathcal{O}(x_i > x_j \mid i < j) = \mathit{false}] = p, \qquad \textit{where } p \textit{ is a constant} < \frac{1}{2}$$

Theorem 1: Feige et. al. [16]: The oracle complexity of top-k under the constant error model is $\Theta(n \log(m/\delta))$, where $m = \min(k, n-k)$. In particular, the oracle complexity of $\max$ is $\Theta(n \log(1/\delta))$.

Tournament Trees. *Tournament trees* (also called *comparison trees* in [14, 13]) are frequently used for the max (min) or selection problems. A tournament tree is a form of max-heap (or min-heap if the minimum is sought) which is a complete binary tree. Every leaf node represents an item and every internal node represents the winner (*e.g.*, the larger item) between two of its children as a result of their comparison. Figures 2a and 2b show two example tournament tree that returns the maximum of $n = 8$ items at the root of the tree, assuming there is no error in the comparisons. For noisy comparisons, at each internal node, multiple comparisons are performed and then the winner is decided, *e.g.*, by a majority vote.



(a) (b) (c)

Figure 2: Constant error model: (a, b) Two tournament trees for max, (c) Comparison tree from [16] after sorting inputs $(2, 5, 7)$.

Max: Consider the $\max$ problem first. In the absence of comparison errors, the max of n items can be found using $n-1$ comparisons along a tournament tree (Figures 2a and 2b). For constant error model, if the probability of error in each comparison is a constant p, it is not hard to see that repeating each comparison in the tournament tree $O(log\frac{n}{\delta})$ times and taking the majority vote to decide the winner will make each comparison right with probability $\leq \frac{\delta}{n}$, and therefore by union bound, the correct max at the root of the tree can be found with probability $\leq \delta$ with total $O(n \log \frac{n}{\delta})$ comparisons.

However, by a simple trick, the better bound of $O(n \log \frac{1}{\delta})$ in Theorem 1 can be obtained: (i) Do comparisons along a balanced tournament tree of height $\log n$; the leaves (in the lowest level) are at level 0, the root is at level $\log n$. (ii) At level $i \in [1, \log n]$, do $i \times O(\log \frac{1}{\delta})$ comparisons for each comparison node. Take majority vote to decide the winner for the higher level. (iii) Output the winner at the root as max. There are $2^{\log n - i} = \frac{n}{2^i}$ internal nodes at level i; therefore the total number of comparisons is $n \times \sum_{\ell=1}^{\log n} \frac{i}{2^i} O(\log \frac{1}{\delta}) = O(n \log \frac{1}{\delta})$. Similarly, by Chernoff bound and union bound[1], and with suitable choices of the constants, the total probability of error that the max item will lose at any of the $\log n$ level is bounded by $\leq \delta$.

Top-k: Next consider the top-k problem and outline the algorithms from [16]. By symmetry, we can assume that $k \leq \frac{n}{2}$, hence we need to show an upper bound of $O(n \log \frac{k}{\delta})$. We consider two cases, (a) when $k \leq \sqrt{n}$, and (b) when $k > \sqrt{n}$.

(a) $k \leq \sqrt{n}$: (i) Build a max-heap using the same algorithm as for max using a tournament tree. However, the error probability is $\frac{\delta}{2k}$, which ensures that the max-heap is consistent with respect to all top-k items with error probability $\leq \frac{\delta}{2}$. (ii) For k times, extract the maximum item from the root, and "reheapify" the max-heap.

[1]Note that the union bound is applied to only the path of the max from the leaves to the root, *i.e.*, to $\log n$ internal nodes and not to $n-1$ internal nodes.

For noiseless comparisons, each of the reheapying steps requires $\log n$ comparisons, *i.e.*, $k \log n$ comparisons in total. By repeating each of the comparisons $O(\log \frac{k \log n}{\delta})$ times, each of the top-k items can be extracted with error probability $\leq \frac{\delta}{2k}$, *i.e.*, with probability $\leq \frac{\delta}{2}$ in total. The total error probability is $\leq \delta$ and the top-k items are output in the sorted order. The number of comparisons in the first step to build the initial heap is $O(n \log \frac{k}{\delta})$. The number of comparisons in the second step to extract the top-k items and to reheapify is $O(k \log n \log(\frac{k \log n}{\delta}))$, which is $O(n \log \frac{k}{\delta})$ for $k \leq \sqrt{n}$.

(b) $k > \sqrt{n}$: We give a sketch of an $O(n \log \frac{n}{\delta})$ algorithm from [16] that can sort n items using $O(n \log \frac{n}{\delta})$ comparisons with probability of error $\leq \delta$, and therefore also solves the top-k problem for $k > \sqrt{n}$ with $O(n \log \frac{k}{\delta})$ comparisons and probability of error $\leq \delta$. The algorithm uses random walk on another form of comparison tree, which are extensions of binary search trees. In a binary search tree, the leaves $x_1 \geq \cdots \geq x_n$ are sorted in order; assuming $x_0 = \infty$ and $x_{n+1} = -\infty$, each leaf $x_i, i \in [1, n+1]$ represents the interval $[x_i, x_{i-1})$. Any internal node with leaves in its subtrees $x_h, x_{h+1}, \cdots, x_\ell$ represents the interval $[x_\ell, x_h)$. When an item x^* is searched on this tree, at each internal node with range $[x_\ell, x_h)$, it is compared with x_z, $z = \lceil \frac{\ell+h}{2} \rceil$. If $x^* \geq x_z$, it is sent to the left child with range $[x_z, x_h)$, otherwise is sent to the right child with range $[x_\ell, x_z)$.

In order to handle noisy comparisons, the comparison tree extends the binary search tree by attaching chains of length $m' = O(\log \frac{n}{\delta})$ to each leaf (in practice they can be implemented as counters from each leaf); each node in a chain has the same interval as that of the corresponding leaf. With noisy comparisons, when an item x^* is searched at a node u with range $[x_\ell, x_h)$ (u is either an internal node or at a chain), first it is checked whether x^* reached at the correct interval, *i.e.*, whether $x^* \geq x_\ell$ and $x^* < x_h$. If these two comparisons succeed, x^* is moved to the correct child of u (its unique child if u is in a chain). Otherwise, it is assumed that some previous step has been wrong, and therefore the search backtracks to the unique parent node of u. By modeling these steps as a Markov process by orienting all edges in the comparison tree toward the right location of x^*. We assume that (without loss of generality), from any node, the probability of transition along the (unique) outgoing edge is $\frac{2}{3}$, and along all the incoming edges together is $\frac{1}{3}$. The search is continued for $m = m_f + m_b < m'$ steps, where m_f, m_b denote the number of forward and backward transitions. When $m = O(\log \frac{n}{\delta})$, $m_f - m_b > \log n$ with high probability. So the final step reaches the correct chain with high probability. If the item does not already appear in the binary search tree, it is inserted. The tree is maintained as a balanced binary search tree by using standard procedures which do not require additional comparisons. By repeating this search followed by insertion procedure for all n items, we get the sorted order in the leaves with $O(n \log \frac{n}{\delta})$ comparisons in total with error probability $\leq \delta$. The matching lower bounds for max and top-k can be found in [16].

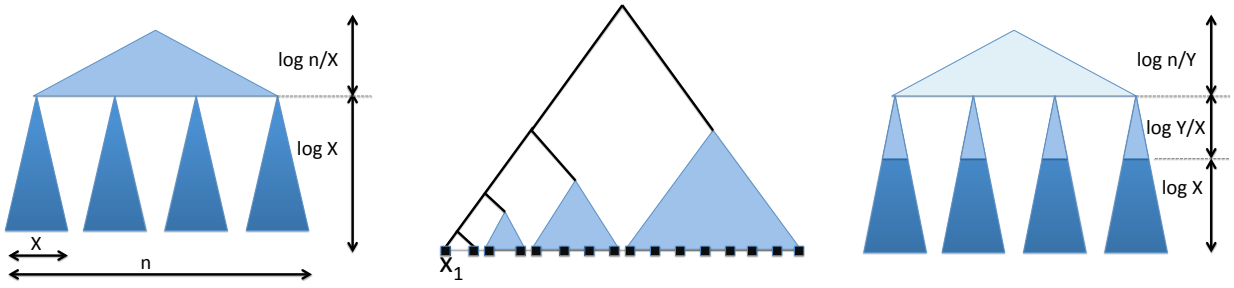
Variable Error Model

Sometimes the comparison error of the noisy oracle may vary with the two input items being considered. As an example, suppose S corresponds to a set of photos of a person, and the items x_i's correspond to the age of the person in the i-th photo. Intuitively, it is much easier to compare the ages in two photos if they are 20 years apart than if they are a few years apart. To model such scenarios Davidson et al. [14, 13] proposed a more general *variable error model*:

$$(\textbf{Variable error model}) \qquad \Pr[\mathcal{O}(x_i > x_j \mid i < j) = false] \leq \frac{1}{f(j-i)}$$

Here $f(x)$ is a *strictly growing error function* that grows with x (*i.e.*, $f = \omega(1)$, *e.g.*, $f(\Delta) = e^\Delta$, Δ, $\sqrt{\Delta}$, $\log \Delta$, $\log \log \Delta$, etc.), is strictly monotone ($f(X) > f(Y)$ for $X > Y$), and satisfies $f(1) > 2$ (the probability of error is always $< \frac{1}{2}$).

Theorem 2: Davidson et al. [14, 13]: For all strictly growing error functions f and constant $\delta > 0$, $n + o(n)$ oracle calls are sufficient to output the maximum item x_1 with error probability $\leq \delta$.



(a) (b) (c)

Figure 3: Variable error model: (a) Upper and lower levels for max, (b) Amplified single X-tree containing max (x_1), (c) Extension to top-k

Further, if $f(\Delta) \geq \Delta$, then $n + O(\log \log n)$ oracle calls are sufficient. If $f(\Delta) \geq 2^{\Delta}$, then $n + O(1)$ oracle calls are sufficient.

Max. The basic idea in [14, 13] was to divide a balanced tournament tree for max (*e.g.*, Figure 2a) into lower and upper levels (Figure 3a). The upper levels will only have $\frac{n}{X} = o(n) \times F(\delta)$ items, for a function F. Then we will apply the tournament algorithm from [16] described above to obtain the max with error probability $\leq \delta$ with $O(\frac{n}{X} \log \frac{1}{\delta}) = o(n)$ comparisons for constant δ, provided the max item x_1 survives in the lower levels.

In the lower levels, *only one* comparison is performed at each internal node, resulting in $< n$ comparisons. Surprisingly, assuming variable error model with any strictly growing error function f, the max item x_1 still survives at the lower levels with probability $\leq 2\delta$ for any constant $\delta > 0$. The leaves of the tournament tree are divided into blocks of length X; they form $\frac{n}{X}$ trees of height $\log X$ at the lower levels, called X*-trees*. We only need to focus on the X-tree containing the max item x_1 (Figure 3b), and ensure that x_1 does not lose in any of the $\log X$ comparisons in this X-tree. The key idea is to do a random permutation of the leaves of the tournament tree. This ensures that the probability of x_1 meeting a large item (say x_2 or x_3) in its first comparison is very small, and therefore using the variable error model, x_1 will have a high probability of winning the comparison. As x_1 progresses in the upper levels, it will have a higher chance of meeting larger items, however, with high probability, the larger items will still appear in the other X-trees, and therefore x_1 will eventually progress to the top of its X-tree despite only one comparison being performed at each node in the lower levels. In particular, we bound the probabilities that (i) P_1 = x_1 meets a large item within a range $x_2, \cdots, x_{\Delta_\ell+1}$ at level ℓ for all $1 \leq \ell \leq \log X$, and (ii) P_2 = x_1 wins the comparison at level ℓ for all $1 \leq \ell \leq \log X$ provided x_1 does not meet $x_2, \cdots, x_{\Delta_\ell+1}$ at level ℓ. We will show that there are choices of X, Δ_ℓ such that both $P_1, P_2 \leq \delta$ and $\frac{n}{X} = o(n)$. This will give an algorithm to find the max with error probability $\leq 3\delta$ using $n + o(n)$ queries to the oracle, which can easily be extended to an algorithm with tolerance δ by choosing constants appropriately.

Instead of showing the bounds in Theorem 2, we will show a weaker upper bound on the number of oracle calls of $n + O(n^{\frac{1}{2}+\mu})$ for max for error functions that are at least linear ($f(\Delta) \geq \Delta$), but will ensure a better tolerance level of $\frac{3\delta}{n^\mu}$, for all constant $\mu < \frac{1}{2}, \delta$. The arguments will be simpler to achieve these bounds, and we will be able to use these bounds to extend this algorithm for max to an algorithm for top-k under the variable error model. The proof of the bounds in Theorem 2 can be found in [14].

(i) First consider P_1. Since the leaves have a random permutation, the probability that any of $x_2, \cdots, x_{\Delta_\ell}$ belongs to the right subtree of an internal node at height ℓ in the path of the max item x_1, by union bounds, is $\leq \frac{\Delta_\ell 2^{\ell-1}}{n-1}$ (these right subtrees are highlighted in Figure 3b). Therefore, $P_1 \leq \sum_{\ell=1}^{\log X} \frac{\Delta_\ell 2^{\ell-1}}{n-1}$. (ii) Next consider P_2. Assuming x_1 does not meet $x_2, \cdots, x_{\Delta_\ell+1}$ at level ℓ, x_1 loses the comparison at level ℓ is $\leq \frac{1}{f(\Delta_\ell)}$ (since f is monotone) $\leq \frac{1}{\Delta_\ell}$. Hence $P_2 \leq \sum_{\ell=1}^{\log X} \frac{1}{\Delta_\ell}$.

We need both P_1, P_2 to be small ($\leq \frac{\delta}{n^\mu}$ for constant $\mu < \frac{1}{2}$); further we need $\frac{n}{X} = o(n)$. Note that P_1 has

Δ_ℓ in the numerator and P_2 has Δ_ℓ in the denominator. Therefore, we need to choose Δ_ℓ carefully to meet our requirements. Here we choose $\Delta_\ell = \frac{2^\ell n^\mu}{\delta}$ and $X = \delta n^{\frac{1}{2}-\mu}$. When the required error probability is a constant δ, and when f has a high growth rate (*e.g.*, exponential functions), better bounds can be obtained. With these choices of Δ_ℓ and X, (i) $P_1 \leq \sum_{\ell=1}^{\log X} \frac{\Delta_\ell 2^{\ell-1}}{n-1} \leq \frac{n^\mu}{2\delta(n-1)} \sum_{\ell=1}^{\log X} 4^\ell = \frac{4n^\mu}{2\delta\times 3(n-1)}(4^{\log X}-1) \leq \frac{n^\mu X^2}{\delta n}$ (when $n \geq 3$) $= \frac{\delta^2 \times n^{1-2\mu}}{\delta n^{1-\mu}} = \frac{\delta}{n^\mu}$. (ii) $P_2 \leq \sum_{\ell=1}^{\log X} \frac{1}{\Delta_\ell} = \sum_{\ell=1}^{\log X} \frac{\delta}{n^\mu 2^\ell} \leq \frac{\delta}{n^\mu}$.

The number of nodes in the upper levels is $\frac{n}{X} = \frac{n}{\delta n^{\frac{1}{2}-\mu}} = \frac{n^{\frac{1}{2}+\mu}}{\delta}$, which is $o(n)$ when $\mu < \frac{1}{2}, \delta$ are constants. The number of oracle calls to obtain the max in the upper levels provided it did not lose in the lower levels with error probability $\leq \frac{\delta}{n^\mu}$ is $O(\frac{n}{X}\log(\frac{n^\mu}{\delta})) = O(n^{\frac{1}{2}+\mu}\log n) = o(n)$ when μ, δ are constants. Hence we compute the max at the root of the tournament tree with at most $n + O(n^{\frac{1}{2}+\mu}\log n) = n + o(n)$ oracle calls and error probability $\leq \frac{3\delta}{n^\mu}$, for all constant $\mu < \frac{1}{2}, \delta$.

Top-k. We can extend the above algorithm to obtain an algorithm for top-k under the variable error model when $k \leq n^\mu$, for constant $\mu < \frac{1}{2}$ and $f(\Delta) \geq \Delta$ with $n + o(n)$ oracle calls and error probability $\leq \delta$ for a constant $\delta > 0$. We again use two levels, upper levels (called *super-upper levels* to avoid confusion) with height $\log(\frac{n}{Y})$ nodes, and lower levels (called *super-lower levels*) with height $\log Y$. In the super-upper levels, the top-k algorithm from [16] is run with error probability $\leq \delta$. The super-lower levels are divided into multiple tournament trees, each with $Y = \frac{\delta n}{k^2}$ leaves. The items are again permuted randomly. First we claim that, when $k \leq n^\mu$ for $\mu < \frac{1}{2}$, then all the top-k items will appear in different tournament trees with error probability $\leq \delta$. This follows from union bound: the probability that any of the tournament trees with Y leaves containing one of the top-k items will contain another top-k item is $\leq \frac{Y(k-1)}{n-1} \leq \frac{kY}{n} = \frac{k\delta n}{nk^2} = \frac{\delta}{k}$. Once again, by union bound, all top-k items appear in different tournament trees with probability $\leq \delta$.

Now in the super-lower levels, every top-k item is the maximum item in its respective tournament tree, and we need to return each of them at the root of its tournament tree with error probability $\leq \frac{\delta}{k}$; then by union bound the error probability is $\leq \delta$ for the lower levels. We use the algorithm for max as described above. However, we need a more careful analysis as otherwise the bound would only guarantee the error probability in the individual tournament tree with Y nodes to be $\leq \frac{\delta}{Y^\mu}$ which may be much larger than $\frac{\delta}{k}$.

We divide each tournament tree in the super-lower levels again into upper and lower levels; the lower levels comprise X-trees (Figure 3c). Select $X = \delta Y^{\frac{1}{2}-\mu}$, but $\Delta_\ell = \frac{2^\ell n^\mu}{\delta}$. Since all n items are randomly permuted among the leaves of all tournament trees (instead of only permuting the Y items within each such tree), (i) Using similar calculations as in the case of max, $P_1 \leq \frac{n^\mu X^2}{\delta n} = \frac{X^2}{\delta n^{1-\mu}} = \frac{\delta^2 Y^{1-2\mu}}{\delta n^{1-\mu}} \leq \frac{\delta^2 n^{1-2\mu}}{\delta n^{1-\mu}} = \frac{\delta}{n^\mu}$ which is at most $\frac{\delta}{k}$ when $k \leq n^\mu$. (ii) Using previous calculations, $P_2 \leq \sum_{\ell=1}^{\log X} \frac{1}{\Delta_\ell} \leq \frac{\delta}{n^\mu} \leq \frac{\delta}{k}$.

In the upper levels of each of the tournament trees, that have only $\frac{Y}{X}$ items (Figure 3c), the max can be found with $O(\frac{Y}{X}\log\frac{k}{\delta})$ oracle calls with error probability $\leq \frac{\delta}{k}$. Hence the total error probability that each of top-k items wins in its respective tournament tree is $\leq 3\delta$. Here $X = \delta Y^{\frac{1}{2}-\mu}$. Hence, the total number of comparisons in the super-lower levels is $\frac{n}{Y} \times (Y + O(\frac{Y}{\delta Y^{\frac{1}{2}-\mu}}\log\frac{k}{\delta})) = n + O(\frac{n}{Y^{\frac{1}{2}-\mu}}\log k)$ for constant δ. Note that $Y^{\frac{1}{2}-\mu} = (\frac{\delta n}{k^2})^{\frac{1}{2}-\mu} \geq (\delta n^{1-2\mu})^{\frac{1}{2}-\mu} \geq n^\epsilon$ for a constant $\epsilon > 0$ (since $k \leq n^\mu$ and $\mu < \frac{1}{2}$ is a constant). Therefore, $O(\frac{n}{Y^{\frac{1}{2}-\mu}}\log k) = o(n)$.

In the upper levels, with $\frac{n}{Y}$ nodes, the number of oracle calls to achieve error probability $\leq \delta$ is $O(\frac{n}{Y}\log\frac{k}{\delta}) = O(\frac{k^2}{\delta}\log\frac{k}{\delta})$, which is $o(n)$ for $k < n^\mu$ and constant $\mu < \frac{1}{2}$. Hence with total error probability $\leq 4\delta$, we find all top-k items with $n + o(n)$ oracle calls, for all $k \leq n^\mu$, constant $\mu < \frac{1}{2}$, when the variable error function is at least linear. Further, the top-k items are returned in sorted order (since the top-k algorithms in [16] return them in sorted order).

Summary: Under the constant error model, to achieve the correct answers with tolerance δ, $O(n \log \frac{1}{\delta})$ oracle calls for max and $O(n \log \frac{k}{\delta})$ oracle calls for top-k, $k \leq \frac{n}{2}$, suffice respectively (and these bounds are tight) [16]. A better bound can be obtained under the variable error model, when the error in comparisons depend on the relative distance of the items in the sorted order. Then only $n + \texttt{smaller order terms}$ suffice for max for constant tolerance δ, and also for top-k for smaller values of k and when the error functions are at least linear [14, 13].

Discussion: The variable as well as the constant error models assume that, given two distinct items, the oracle will always return the correct answer with probability $> \frac{1}{2}$. However, this may not always hold in crowdsourcing. For instance, if two photos of a person are taken a few minutes (even a few days) apart, it will be almost impossible for any human being to compare their time by looking at them. However, both variable and constant error models cannot support such scenarios. Such an error model has been considered by Ajtai et. al. [4] who assume that, if the values of two items being compared differ by at least Δ for some $\Delta > 0$, then the comparison will be made correctly. When the two items have values that are within Δ, the outcome of the comparison is unpredictable. Of course, it may be impossible to compute the correct maximum under this model for certain inputs. However, the authors show that the maximum can be obtained within a 2Δ-additive error with $O(n^{3/2})$ comparisons and $\Omega(n^{4/3})$ comparisons are necessary (they also generalize these upper and lower bounds). In contrast, even under the constant error model, the correct maximum can be computed with high probability with $O(n)$ comparisons.

Open question: What are the upper and lower bounds on the oracle complexity and guarantee on the accuracy level if we combine the constant/variable error model with the model in [4], where the answers can be arbitrary if the two items being compared have very similar values?

2.2 Concave Cost Functions

We have used the fixed-cost model so far in which each question incurs unit cost, resulting in a total cost which is the number of comparisons performed. However, in some scenarios, many questions are asked together, *i.e.*, multiple tasks are grouped in *batches* before they are assigned to a worker[2]. In these cases, the total payment for all these questions in a batch can be less than the payment when the questions are asked one by one, since it is likely to require less effort from the worker (for instance, the worker does not need to submit an answer before getting the next question and potentially can choose the answers faster). Such cost functions can be modeled as non-negative monotone *concave functions*[14], *i.e.*, (**Concave functions**) *for all* $t \in [0, 1], N_1, N_2$,

$$g(tN_1 + (1-t)N_2) \geq tg(N_1) + (1-t)g(N_2).$$

Examples include $g(N) = N^a$ (for a constant a, $0 < a \leq 1$), $\log N, \log^2 N, \log \log N$, $2^{\sqrt{\log N}}$, etc. Non-negative concave functions are interesting since they exhibit the *sub-additive property*:

$$g(N_1) + g(N_2) \geq g(N_1 + N_2).$$

Concave cost functions display a tension between the number of rounds and the total number of questions asked in an algorithm since (1) it is better to ask many questions in the same batch of the oracle in the same round, rather than distributing them to many different batches and multiple rounds; but (2) the cost function is monotone, so we cannot ask too many questions as well. As an extreme (and non-realistic) example, for finding the maximum item under the constant concave cost function $g(N) = a$, $a > 0$, we could ask all $\binom{n}{2}$ comparisons of the oracle in the same batch and compute the maximum; however, this is not a good strategy for a linear cost function. We assume that the cost function g is known in advance, and the goal is to develop algorithms for arbitrary concave cost functions that aim to minimize the total incurred cost.

[2] *Batches* denote grouping comparison tasks before assigning them to a worker, while after each *round* the machine gets backs the answers from the workers in the batches of that round.

If the oracle receives a set with N comparisons in the same batch, we incur a cost of $g(N)$ for these N comparisons. We still assume that the answers to two different comparisons (where at least one item is different) are independent even in the same batch. However, we cannot ask the oracle to compare the same pair of items in the same batch when we need redundancy for erroneous answer (*i.e.*, the same worker cannot be asked to compare the same two items twice). In this case, multiple comparisons of the same pair of items must go to different batches, and therefore will incur additional cost. We also assume that there is no limit on the maximum size of the batch of questions that can be asked of the oracle at once. Unlike the fixed-cost model where it sufficed to minimize the total number of comparisons, now our goal is to optimize the total cost under g.

Algorithms. Assuming no comparison errors, [14] gives a simple $O(\log\log n)$-approximation algorithm[3] to find the max item using, once again, a tournament tree (a similar algorithm is given in [19] for finding max in $O(\log\log n)$ rounds with $O(n)$ comparisons in total). However, unlike the standard tournament tree, at any level h of the tree with B_h nodes, all possible $\binom{B_h}{2}$ comparisons are performed. In addition, the internal nodes at level h have 2^h children from the lower level $h-1$. It is shown that, the number of levels of the tree is $\log\log n$, and at any level of the tree, at most n comparisons are performed incurring cost $g(n)$, giving an algorithm with cost $O(g(n)\log\log n)$. Since the cost of the optimal algorithm has a lower bound $OPT \geq g(n-1) \geq g(n) - g(1)$. Assuming $g(1)$ to be a constant, this gives an $O(\log\log n)$-approximation algorithm. The same algorithm can be extended for the constant error model by repeating the comparisons multiple times in different batches of calls to the oracle[4]. Two other algorithms for finding max in $O(\log\log n)$ rounds are also mentioned in [14] that can be extended to give the same approximation: by Valiant [41] and by Pippenger [36]; however, these algorithms are more complex to implement for practical purposes. Nevertheless, the algorithm based on Pippenger's approach can be extended to a randomized algorithm for top-k under the concave cost function and no comparison error, with an expected cost of OPT $\times\, O(\log\log n)$.

Summary: Assuming no comparison error, we get an $O(\log\log n)$-approximation for max under any concave cost function g. For the constant error model, the same algorithm with repeated comparisons gives an $O(\log n)$-approximation for any concave cost function g, and $O(\log\log n)$-approximation for $g(n) = n^\alpha$, where $\alpha \in (0, 1]$ is a constant. Using standard and known techniques, an $O(\log\log n)$-approximation can be achieved for top-k and no error (however, not simple to implement for practical purposes).

Discussions: For unit cost function, assigning tasks to single or multiple workers amounts the same total cost, which is not the case for concave cost functions. In some practical crowd-sourced applications, it may be useful to assume an upper bound B on the batch size, as workers may not be able to answer questions in a very large batch without error due to fatigue or time constraint. Further, the independence assumption is even less likely to hold when multiple questions are asked of the oracle in the same batch. Finding a practical model when tasks are grouped into batches is a direction to explore.

Open questions: (1) Assuming no comparison errors, can we devise algorithms for concave cost functions that have better bounds than the algorithms in [14] (for max), and are simpler to implement (for top-k)? What can we infer about optimal algorithms? Can we have a better lower bounds for OPT than $g(n-1)$ for all or some concave cost functions g?
(2) The bounds for the constant error model have been obtained in [14] under concave cost functions simply by repeating individual comparisons by different workers and taking the majority vote. How can we obtain algorithms with better bounds for constant and variable error models under arbitrary concave cost functions?

[3] An algorithm is a $\mu(n)$-*approximation algorithm* for some non-decreasing function μ, if for every input of size n it can find the solution (*e.g.*, the maximum item) with a cost $\leq \mu(n) \times$ OPT.

[4] For constant δ, $O(\log\log n)$-approximation when $g(n) = n^\alpha$, α is a constant $\in (0, 1]$, and $O(\log n)$-approximation for arbitrary concave function g

3 Skyline queries

Skylines. Let S be a set of n items. We assume these items admit a full (but not necessarily strict) order $\leq_i$ along d dimensions $i \in \{1, \ldots, d\}$. We also write $v \preccurlyeq v'$ to denote that $v \leq_i v'$ for each $i \leq d$. When $v \preceq v'$ and there is some $i \leq d$ such that $v <_i v'$, we say that v' *dominates* v, which we denote by $v \prec v'$.

Given a set of d-dimensional items S, the *skyline* of S is the set of items that are not dominated (we assume that two items can not coincide):

$$\text{Sky}(S) = \{v \in S \mid \forall v' \in S \setminus \{v\}, \exists i \leq d.\ v >_i v'\}.$$

Input: A set of items S; tolerance δ; noisy comparison oracle $\mathcal{O}$. **Objective:** Return $\text{Sky}(S)$ with error probability δ.

In our example of Figure 1, the skyline consists of locations $\{a, b, c, d, e\}$, whereas location h, for instance, is dominated by location a.

Algorithms

Skyline queries must identify the items that are not dominated. The classical skyline algorithms are usually based on one of two paradigms (possibly combined): divide and conquer, or sweeping line approaches. In the divide and conquer approach, input items are split around the median on one of the dimensions, then the skyline of both halves are computed. After the splitting, all items from the smaller half that are dominated by some larger item are pruned out. Finally the union of the two partial results is returned. One major stumbling block with this divide and conquer approach for skylines in presence of noise seems to be the issue of splitting items around the median; with noisy comparisons, identifying exactly the set of items which are larger than the median is as expensive as a full sort of the input [16]. Because of this it seems unlikely that the classical divide and conquer skyline algorithms [23, 24] can be adapted into efficient algorithms in presence of noisy comparisons, though more intricate schemes as proposed for parallel sort with noise [28] might still provide efficient algorithms with noisy comparisons.

In contrast, the sweeping line approach adapts easily to noisy comparisons. To compute iteratively the skyline, each iteration adds to the result the maximal item for lexicographic order (the maximum on $<_1$, if there are ties we take the maximum for $<_2$ among those, etc.) among the items that are not dominated. The algorithm stops after k iterations.

We first present efficient procedures to check if a given item is dominated. Given an item v and a set C of items, we say that C dominates v if there is $v' \in C$ such that $v \prec v'$, i.e., if $\bigvee_{v' \in C} \bigwedge_{j \leq d} v <_j v'$. The formula expressing domination is thus a composition of boolean functions involving basic comparisons, which can be computed in $O(d|C| \log \frac{1}{\delta})$ [20, 31]. Another approach to check dominance consists in sorting the set of items C along each dimension, then using binary search to obtain the relative position v within C on each dimension. The complexity of sorting C is $O(d|C| \log \frac{d|C|}{\delta})$. Once C is sorted, each item v can be inserted along each dimension with error probability δ/d in $O(\log \frac{d|C|}{\delta})$.

We next observe that classical algorithms for MAX with noisy comparisons (and similarly with boolean formulae) are what we call "trust-preserving". This means that these algorithms return in $O(n)$ the correct output with (at most) the same error probability as the input oracle. Our line-sweeping algorithms thus compute one skyline point per iteration, using any of the two dominance-checking procedures to compute the maximal item among those that are not dominated. The algorithm makes sure the probability of error per iteration is at most δ/k. The value of k is not known in advance but the error requirements are met by adapting a classical

trick in output sensitive algorithms [9] and proceeding with maximal error $\delta/2^i$ as soon as 2^{2^i} items have been discovered.

As a result, efficient upper bounds can be obtained for skyline computation:

Theorem 3: Groz and Milo [20]: $\text{Sky}(S)$ can be computed with any of the following complexities:

1. $O(dn \log(dn/\delta))$
2. $O(dk^2 n \log(k/\delta))$
3. $O(dkn \log(dk/\delta))$

The first result corresponds to sorting all input items on every dimension. The second and third compute the skyline iteratively; each iteration adding the lexicographically maximal item among those that are not dominated. The second uses the boolean formula approach to check dominance whereas the third sorts the skyline item as they are discovered and then uses binary insertion to check dominance. The first and third results only count the number of calls to the comparison oracles executed, so that computational complexity may be higher: sorting the input provides all the ordering information necessary to compute the skyline, but a classical algorithm to compute the skyline (without noise) must still be applied before a skyline can be returned based on these orderings. The exact complexity of skylines in this model remains open since the only lower bound we are aware of is $\Omega(n \log(k/\delta) + dn)$. But even without noise an optimal bound of $\Theta(n \log k)$ is only known when d is assumed to be constant.

Rounds. The number of rounds required by those algorithms are respectively $O(\log n)$, $O(k \cdot \log\log n \cdot \log^*(d) \cdot \log^*(k))$, and $O(k \cdot \log\log n \cdot \log k)$. Two interesting patterns can be observed. First the number of rounds does not depend on δ: the additional oracle calls that must be performed to gain accuracy can always be processed in parallel. Second, none of the three algorithms proposed so far for skyline computation stands out in terms of rounds: each can outperform the others depending on input parameters. Last, we recall from [41] that $\log\log n$ rounds are required to compute the maximum of n items with $O(n)$ comparisons even when comparisons are accurate, therefore it is not surprising that all of our algorithms require at least $\log\log n$ rounds even for small values of k.

Summary: [20] presents several algorithms to compute skylines under the constant error model. The complexity of these algorithms is dn multiplied (depending on the algorithm) by k or k^2, and logarithms in d, n, or $1/\delta$. Each algorithm may outperform the others for some values of the input parameters both in terms of the cost and the number of rounds.

Discussion: Skylines are generally harder than sorting problems, since even in the case of two dimensions the comparisons required to check if the skyline contains all items necessarily provide a complete ordering of all items. Therefore one cannot hope to achieve drastic improvements in the constant factors as we showed for Max and Top-k, nevertheless one could reasonably hope that variable error models could benefit the algorithms in a sense that remains to be defined.

Open questions: (1) Can the bounds of the skyline problem be improved assuming other error models (*e.g.*, the variable error model)?
(2) How many comparisons are required to compute the skyline when neither d nor k are assumed to be constant? This question is already open without noise in comparisons [10]. To what extend do noisy comparisons raise the complexity?
(3) Multiple variants of skyline problems such as layers of skylines or approximated skylines have been considered in standard data models without noise. It could be interesting to investigate how such problems can be tackled by the Crowd. The approximation of skylines, in particular, can be justified by the observation that the skyline cardinality tends to increase sharply with the number of criteria considered, on random instances.

4 Related work

In the past few years, crowdsourcing has emerged as a topic of interest in the database and other research communities in Computer Science. Here we mention some of the relevant work in crowd sourcing, and computation with noisy operations in general.

Crowdsourcing database/data mining operators. Different crowd-sourced databases like CrowdDB [17], Deco [32], Qurk [29], AskIt![6] help users decide which questions should be asked of the crowd, taking into account various requirements such as latency, monetary cost, quality, etc. On the other hand, several papers focus on a specific operator, like joins [44], group-by or clustering[18, 46], entity resolution [43, 45], filters [34, 33], frequent patterns identification [5], centroid [22], and investigate techniques to optimize the execution of this operator in a crowdsourcing environment.

Max, top-k, and ranking. Multiple approaches have been proposed to sort or search the maximal items using the crowd. The comparative merits of comparisons and ratings have thus been evaluated experimentally in the Qurk system [29], as well as the effect of implementation parameters such as batch size. Ranking features were similarly proposed in the CrowdDB system [17]. Heuristic to compute the maximal item with the Crowd have been introduced in [42]. In their model, the Crowd must return the largest item in the batch, and the probability of returning each item depends on its rank in the batch. This line of work thus also allows to model distance-dependent error, though not in a comparison framework. A classical issue in voting theory is rank aggregation: given a set of ranking, one must compute the "best" consensus ranking. This is the approach for the *judgment problem* studied in [21] (given a set of comparison results, which item has the maximum likelihood of being the maximum item) and the next vote problem (given a set of results, which future comparisons will be most effective). They propose hardness results and heuristics to compute the most likely maximum. Their model assumes a comparison oracle with constant error probability, and the input consists in a matrix summarizing the results of oracle queries that have already been performed. Beyond the computation of the most likely maximum they also consider the problem of best using a fixed budget of additional questions that should be issued to complement the vote matrix. In [37], the authors consider the crowdsourced top-k problem when the unit of task (for time and cost) is not comparing two items, but ranking any number of items. The error model is captured by potentially different rankings of the same items by different people. If many people disagree on the ranking of certain items, more people are asked to rank these items to resolve the conflict in future rounds. A related problem studied in the machine learning community is that of learning to rank in information retrieval (*e.g.*, [25, 8, 38]). Our noisy comparison model with constant error probability appears with some refinements in [16, 19, 31, 28], while other models have also been considered for ranking problems in [35] (the total number of erroneous answers in bounded) and [4] (comparisons are arbitrary if two items are too close).

Skyline. While Pareto-optimality has received ample attention in machine learning, algorithm [23, 10] and traditional database [7] communities, fewer uncertain data models have been considered so far for skylines in the database community. One popular model considers inputs given by a (discrete) probability distribution on the location of items in space [2]. Heuristic approaches have been considered over incomplete or imprecise data [26, 27]. Beside [20], the parallel computation of skylines has only been considered in the absence of noise [1].

Number of query rounds and beyond worst case. The number of rounds required by fault-tolerant sorting and searching algorithms has been a topical issue for a long time [16], in parts driven by parallelism in circuits. Efficient Sorting [28] and Maxima [19] algorithms with respectively $\log n$ and $\log \log n$ rounds have thus been proposed, matching the theoretical lower bounds. Whether in our error model or others, worst-case complexity may appear too conservative a measure to derive efficient crowdsourcing algorithms, so it may be interesting to look at average complexity or instance-optimality, which have been considered for top-k [15] and skyline [3] problems in other models.

5 Discussions and Future Work

The crowd has become a valuable resource today to answer queries on which the computers lack sufficient expertise. This article gives an overview of known theoretical bounds under different cost and error models for max/top-k and skyline queries. These results assume a simple probabilistic model for the crowd as a noisy oracle performing pairwise comparisons between elements. This simple model for the crowd lays the foundation for several interesting future research directions for practical purposes:

- The actual error, cost, and latency models for the crowd, as mentioned earlier, are hard to formalize and need to be substantiated with real experiments with crowd that model their behavior in different scenarios.
- None of the algorithms in this article considers the actual number of crowd workers who are involved in the tasks, as long as there are sufficient number of crowd workers for redundant comparisons. One can study the dependency on the cost. accuracy, and latency of the algorithms for different numbers of crowd workers involved, and variations of the independence assumptions and uniformity of error models across all workers.
- Even assuming the model is a reasonable approximation of the crowd's behavior, the algorithms must know the parameters (tolerance δ, error function f) precisely. Estimating those parameters accurately with low cost may require additional experimental study.
- The relevance of asymptotic estimations provides some hindsight on the cost one may expect to achieve, but may not accurately capture the cost for practical purposes where either more comparisons may be needed or fewer comparisons may suffice for reasonable results. Similarly, some easier comparisons can be performed by machines to save on cost and time. A rigorous analysis of human behaviors with experiments on crowd sourcing platforms, and adapting the models and algorithms accordingly, will be an important and challenging research direction.

References

[1] F. N. Afrati, P. Koutris, D. Suciu, and J. D. Ullman. Parallel skyline queries. In *ICDT*, pages 274–284, 2012.

[2] P. Afshani, P. K. Agarwal, L. Arge, K. G. Larsen, and J. M. Phillips. (approximate) uncertain skylines. *Theory Comput. Syst.*, 52(3):342–366, 2013.

[3] P. Afshani, J. Barbay, and T. M. Chan. Instance-optimal geometric algorithms. In *FOCS*, pages 129–138, 2009.

[4] M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. In *ICALP (1)*, pages 37–48, 2009.

[5] Y. Amsterdamer, Y. Grossman, T. Milo, and P. Senellart. Crowd mining. In *SIGMOD*, pages 241–252, 2013.

[6] R. Boim, O. Greenshpan, T. Milo, S. Novgorodov, N. Polyzotis, and W.-C. Tan. Asking the right questions in crowd data sourcing. *ICDE*, 0:1261–1264, 2012.

[7] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[8] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML*, ICML '05, pages 89–96, 2005.

[9] T. M. Chan. Optimal output-sensitive convex hull algorithms in two and three dimensions. *Discrete & Computational Geometry*, 16(4):361–368, 1996.

[10] T. M. Chan and P. Lee. On constant factors in comparison-based geometric algorithms and data structures. In *SOCG*, page 40, 2014.

[11] ClickWorker. In `http://www.clickworker.com`.

[12] CrowdFlower. In `http://www.crowdflower.com`.

[13] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Using the crowd for top-k and group-by queries. In *ICDT*, pages 225–236, 2013.

[14] S. B. Davidson, S. Khanna, T. Milo, and S. Roy. Top-k and clustering with noisy comparisons. *ACM Trans. Database Syst. (TODS)*, 39(4):35:1–35:39, 2014.

[15] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[16] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM J. Comput.*, 23(5):1001–1018, Oct. 1994.

[17] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. Crowddb: answering queries with crowdsourcing. In *SIGMOD*, pages 61–72, 2011.

[18] R. Gomes, P. Welinder, A. Krause, and P. Perona. Crowdclustering. In *NIPS*, pages 558–566, 2011.

[19] N. Goyal and M. Saks. Rounds vs. queries tradeoff in noisy computation. *Theory of Computing*, 6(1):113–134, 2010.

[20] B. Groz and T. Milo. Skyline queries with noisy comparisons. In *PODS*, pages 185–198, 2015.

[21] S. Guo, A. Parameswaran, and H. Garcia-Molina. So who won?: dynamic max discovery with the crowd. In *SIGMOD*, pages 385–396, 2012.

[22] H. Heikinheimo and A. Ukkonen. The crowd-median algorithm. In *HCOMP*, 2013.

[23] D. G. Kirkpatrick and R. Seidel. Output-size sensitive algorithms for finding maximal vectors. In *SOCG*, pages 89–96, 1985.

[24] H. T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *J. ACM*, 22(4):469–476, 1975.

[25] T.-Y. Liu. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, Mar. 2009.

[26] C. Lofi, K. E. Maarry, and W.-T. Balke. Skyline queries in crowd-enabled databases. In *EDBT*, pages 465–476, 2013.

[27] C. Lofi, K. E. Maarry, and W.-T. Balke. Skyline queries over incomplete data - error models for focused crowd-sourcing. In *ER*, pages 298–312, 2013.

[28] Y. Ma. An $o(n \log n)$-size fault-tolerant sorting network (extended abstract). In *STOC*, pages 266–275, 1996.

[29] A. Marcus, M. S. Bernstein, O. Badar, D. R. Karger, S. Madden, and R. C. Miller. Twitinfo: aggregating and visualizing microblogs for event exploration. In *CHI*, pages 227–236, 2011.

[30] MicroWorkers. In `https://microworkers.com`.

[31] I. Newman. Computing in fault tolerant broadcast networks and noisy decision trees. *Random Struct. Algorithms*, 34(4):478–501, 2009.

[32] A. Parameswaran, H. Park, H. Garcia-Molina, N. Polyzotis, and J. Widom. Deco: Declarative crowdsourcing. Technical report, Stanford University.

[33] A. G. Parameswaran, S. Boyd, H. Garcia-Molina, A. Gupta, N. Polyzotis, and J. Widom. Optimal crowd-powered rating and filtering algorithms. *PVLDB*, 7(9):685–696, 2014.

[34] A. G. Parameswaran, H. Garcia-Molina, H. Park, N. Polyzotis, A. Ramesh, and J. Widom. Crowdscreen: algorithms for filtering data with humans. In *SIGMOD*, pages 361–372, 2012.

[35] A. Pelc. Searching games with errors - fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1-2):71–109, 2002.

[36] N. Pippenger. Sorting and selecting in rounds. *SIAM J. Comput.*, 16(6):1032–1038, Dec. 1987.

[37] V. Polychronopoulos, L. de Alfaro, J. Davis, H. Garcia-Molina, and N. Polyzotis. Human-powered top-k lists. In *WebDB*, pages 25–30, 2013.

[38] F. Radlinski and T. Joachims. Active exploration for learning rankings from clickthrough data. In *SIGKDD*, KDD '07, pages 570–579, New York, NY, USA, 2007. ACM.

[39] SamaSource. In `http://www.samasource.org`.

[40] A. M. Turk. In `https://www.mturk.com/`.

[41] L. G. Valiant. Parallelism in comparison problems. *SIAM J. Comput.*, 4(3):348–355, 1975.

[42] P. Venetis, H. Garcia-Molina, K. Huang, and N. Polyzotis. Max algorithms in crowdsourcing environments. In *WWW*, pages 989–998, 2012.

[43] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[44] J. Wang, G. Li, T. Kraska, M. J. Franklin, and J. Feng. Leveraging transitive relations for crowdsourced joins. In *SIGMOD*, pages 229–240, 2013.

[45] S. E. Whang, P. Lofgren, and H. Garcia-Molina. Question selection for crowd entity resolution. *Proc. VLDB Endow.*, 6(6):349–360, Apr. 2013.

[46] J. Yi, R. Jin, A. K. Jain, S. Jain, and T. Yang. Semi-crowdsourced clustering: Generalizing crowd labeling by robust distance metric learning. In *NIPS*, pages 1781–1789, 2012.

SampleClean: Fast and Reliable Analytics on Dirty Data

Sanjay Krishnan[1], Jiannan Wang[1], Michael J. Franklin[1], Ken Goldberg[1],
Tim Kraska[2], Tova Milo[3], Eugene Wu[4]

[1]UC Berkeley, [2]Brown University, [3]Tel Aviv University, [4]Columnia University

Abstract

An important obstacle to accurate data analytics is dirty data in the form of missing, duplicate, incorrect, or inconsistent values. In the SampleClean project, we have developed a new suite of techniques to estimate the results of queries when only a sample of data can be cleaned. Some forms of data corruption, such as duplication, can affect sampling probabilities, and thus, new techniques have to be designed to ensure correctness of the approximate query results. We first describe our initial project on computing statistically bounded estimates of `sum`*,* `count`*, and* `avg` *queries from samples of cleaned data. We subsequently explored how the same techniques could apply to other problems in database research, namely, materialized view maintenance. To avoid expensive incremental maintenance, we maintain only a sample of rows in a view, and then leverage SampleClean to approximate aggregate query results. Finally, we describe our work on a gradient-descent algorithm that extends the key ideas to the increasingly common Machine Learning-based analytics.*

1 Introduction

Data are susceptible to various forms of corruption such as missing, incorrect, or inconsistent representations [42]. Real-world data are commonly integrated from multiple sources, and the integration process may lead to a variety of errors [20]. Data analysts report that data cleaning remains one of the most time consuming steps in the analysis process [3]. Identifying and fixing data error often requires manually inspecting data, which can quickly become costly and time-consuming. While crowdsourcing is an increasingly viable option for correcting some types of errors [29, 22, 48, 24, 38, 4, 14], it comes at the significant cost of additional latency and the overhead of managing human workers.

On the other hand, ignoring the effects of dirty data is potentially dangerous. Analysts have to choose between facing the cost of data cleaning or coping with consequences of unknown inaccuracy. In this article, we describe a middle ground that we call SampleClean. SampleClean leverages insights from statistical estimation theory, approximate query processing, and data cleaning to devise algorithms for estimating query results when only a sample of data is cleaned. The intriguing part of SampleClean is that results computed using a small clean sample, results can be more accurate than those computed over the full data due to the data cleaning. This approximation error is boundable, unlike the unknown data error, and the tightness of the bound is parametrized by a flexible cleaning cost (i.e., the sampling size).

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

The motivation behind SampleClean is in many ways analogous to that of Approximate Query Processing (AQP) [36, 15, 23, 6]. For decision problems, exploratory analysis problems, and visualization, it often suffices to return an approximate query result bounded in confidence intervals. For many common aggregates, a sample size of k results in an approximation error $O(\frac{1}{\sqrt{k}})$, and therefore every additional ϵ factor of accuracy costs quadratically more. In applications where approximation can be tolerated, sampling avoids the expensive "last mile" of processing and timely answers facilitate improved user experiences and faster analysis.

In traditional AQP, approximation necessarily sacrifices accuracy for reduced latency. However, the goal of SampleClean differs from AQP, as SampleClean trades off data cleaning cost for gradual improvements in query accuracy. While SampleClean introduces approximation error, the data cleaning mitigates bias due to dirty data. There is a break-even point where a sufficient amount of data is cleaned to facilitate an accurate approximation of queries on the cleaned data, and in this sense, sampling actually improves the accuracy of the query result.

SampleClean [45] and all of its extensions [30, 31, 25], work in the *budgeted data cleaning* setting. An analyst is allowed to apply an expensive data transformation $C(\cdot)$ to only $k \ll N$ rows in a relation. One solution could be to draw k records uniformly at random and apply data cleaning, e.g., a direct extension of AQP [6] to the cleaned sample. However, data cleaning presents a number of statistical methodology problems that make this hard. First, $C(\cdot)$ may change the sampling statistics, for example, duplicated records are more likely to be sampled. Next, query processing on partially clean data, i.e., a mix of dirty and clean data, can lead to unreliable results due to the well known Simpsons Paradox. Finally, high-dimensional analytics such as Machine Learning may be very sensitive to sample size, perhaps even more so than to dirty data, and techniques for avoiding sample size dependence are required. Our research contribution in SampleClean is to study estimation techniques that avoid or mitigate these challenges.

There are two contrasting estimation techniques to address every budgeted data cleaning problem: *direct* estimation and *correction*. The direct estimate applies a query, possibly with some re-weighting and scaling to account for data cleaning, to the cleaned sample of data. Alternatively, a sample of cleaned data can also be used to correct the error in a query result over the dirty data. There is an interesting theoretical tradeoff between these approaches, where the direct approach is *robust* as its accuracy is independent of the magnitude of data error, and the correction is *sample-efficient* as its accuracy is less dependent on sample size than the direct estimate.

There are a number of new research opportunities at the intersection of data cleaning and approximate query processing. We applied the same principles to other domains with expensive data transformations such as Materialized View Maintenance and Machine Learning. In this article, we highlight three projects:

SampleClean [45] [1]: SampleClean estimates aggregate (`sum`, `count`, `avg`) queries using samples of clean data. SampleClean reweights the data to compensate for changes in sampling statistics such that query result estimations remain unbiased and bounded in confidence intervals.

View Cleaning [30]: View Cleaning generalizes the notion of data cleaning to include expensive incremental maintenance of out-of-date views. Staleness in materialized views (MVs) manifests itself as data error, i.e., a stale view has missing, superfluous, and incorrect rows. Like data cleaning, eager MV maintenance is expensive. Aggregate queries are approximated from a stale MV using a small sample of up-to-date data, resulting in bounded estimates.

ActiveClean [31]: ActiveClean extends SampleClean to a class of analytics problems called Convex Data Analytics; subsuming the aggregates studied in SampleClean and including Machine Learning such as Support Vector Machines and Linear Regression. ActiveClean exploits the convex structure of the problem to prioritize cleaning data that is likely to affect the model. ActiveClean directly integrates cleaning into the model training loop and as a result gives a bounded approximation for a given cleaning budget.

This article is organized as follows. Section 2 introduces the project and its main ideas. Section 4/5/6

[1] SampleClean refers to both the entire project and our initial publication.

describes SampleClean, View Cleaning, and ActiveClean respectively. Section 7 reviews the related work in this field. In Section 8, we highlight some of the open problems and future directions of the SampleClean project. Finally, we conclude in Section 9.

2 Background and Main Ideas

This section describes the key idea of SampleClean, namely, that data cleaning can be integrated with approximate query processing leading to bounded approximations of clean query results for a fraction of the cleaning cost.

2.1 Traditional Approximate Query Processing

A number of approximation schemes have been proposed including using Sampling, Wavelets, Sketching, and Hashing (see Cormode et al. for a survey [16]). This article focuses on Sampling-based approximations and we will use the term AQP to refer to such systems (e.g., BlinkDB[6]). Sampling-based approximate query processing is a powerful technique that allows for fast approximate results on large datasets. It has been well studied in the database community since the 1990s [27, 5, 36, 35], and methods such as BlinkDB [6] have drawn renewed attention in recent big data research. An important aspect of this work is confidence intervals, as many types of aggregates can be bounded with techniques such as concentration inequalities (e.g., Hoeffding bounds), large-deviation inequalities (e.g., Central Limit Theorem), or empirically (e.g., Bootstrap). Suppose, there is a relation R and a uniform sample S. AQP applies a query Q to S (possibly with some scaling c) to return an estimate:

$$Q(R) \approx est = c \cdot Q(S)$$

Traditionally, AQP sacrifices accuracy due to sampling for improved query latency. However in AQP, the bounds on est assume that the only source of error is approximation error introduced by sampling, however, the data itself may contain errors which could also affect query results. When the data itself is erroneous, a query result on the full data–let alone a sample, will be incorrect. The main argument for SampleClean is that when data errors significantly affect query results, sampling can be combined with data cleaning to actually improve accuracy. This leads to a counter-intuitive result where it is possible that a query on a cleaned sample of data is more accurate than a query on the entire dirty data.

2.2 Approximate Query Processing on Dirty Data

2.2.1 Two Sources of Errors: Sampling Error and Data Error

If R is dirty, then there is a true relation R_{clean}.

$$Q(R_{clean}) \neq Q(R) \approx est = c \cdot Q(S)$$

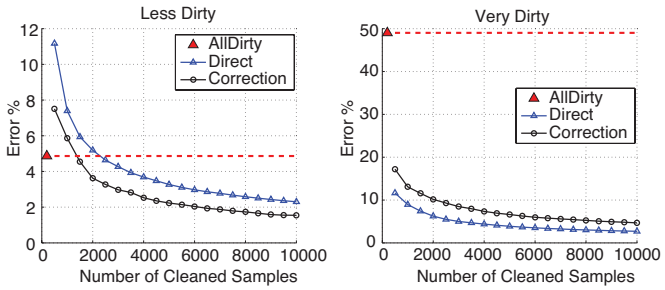
The error in est has two components: error due to sampling ϵ_s and error due to the difference with the cleaned relation $\epsilon_c = Q(R_{clean}) - Q(R)$:

$$\mid Q(R_{clean}) - est \mid \leq \epsilon_s + \epsilon_c$$

While they are both forms of query result error, ϵ_s and ϵ_c are very different quantities. ϵ_s is a random variable due to the sampling, and different samples would result in different realizations of ϵ_s. As a random variable introduced by sampling, ϵ_s can be bounded by a variety of techniques as a function of the sample size. On the other hand, ϵ_c is deterministic, and by definition is an unknown quantity until all the data is cleaned. Thus, the bounds returned by a typical AQP framework on dirty data would neglect ϵ_c.

It is possible that $R_{clean} \neq R$ but $\epsilon_c = 0$. Consider a `sum` query on the relation $R(a)$, where a is a numerical attribute. If half of the rows in R are corrupted with $+1$ and the other half are corrupted with -1, then

Figure 1: Comparison of the convergence of the methods on two TPC-H datasets of 6M tuples with simulated errors 50% error and 5% error. On the dataset with larger errors, the direct estimate gives a narrower confidence interval, and on the other the correction is more accurate.



$Q(R_{clean}) = Q(R)$. The interesting problem is when there are *systematic errors*[43] i.e., $\mid \epsilon_c \mid > 0$. In other words, the corruption that is correlated with the data, e.g., where every record is corrupted with a $+1$.

2.2.2 Key Idea I: Direct Estimate vs. Correction

The key quantity of interest is ϵ_c, and to be able to bound a query result on dirty data, requires that ϵ_c is 0 or bound ϵ_c.

Direct Estimate: This technique is a direct extension of AQP to handle data cleaning. A set of k rows is sampled uniformly at random from the dirty relation R resulting in a sample S. Data cleaning is applied to the sample S resulting in S_{clean}. Data cleaning and sampling may change the statistical and scaling properties of the query Q, so Q may have to be re-written to a query $\widehat{Q}$. $\widehat{Q}$ is applied to the sample S_{clean} and the result is returned. There are a couple of important points to note about this techniques. First, as in AQP, the direct estimate only processes a sample of data. Next, since it processes a cleaned sample of data, at no point is there a dependence on the dirty data. As we will show later in the article, the direct estimate returns a result whose accuracy is independent of the magnitude or rate of data error. One way to think about this technique is that it ensures $\epsilon_c = 0$ within the sample.

Correction: The direct estimate suffers a subtle drawback. Suppose, there are relatively few errors in the data. The errors introduced by sampling may dominate any error reductions due to data cleaning. As an alternative, we can try to estimate ϵ_c. A set of k rows is sampled uniformly at random from the dirty relation R resulting in a sample S. Data cleaning is applied to the sample S resulting in S_{clean}. The difference in applying $\widehat{Q}$ to S and $\widehat{Q}$ to S_{clean} gives an estimate of ϵ_c. The interpretation of this estimate is a correction to the query result on the full dirty data. In contrast to the direct estimate, this technique requires processing the entire dirty data (but only cleaning a sample). However, as we will later show, if errors are rare this technique gives significantly improved accuracy over the direct estimates.

2.2.3 Key Idea II: Sampling to Improve Accuracy

Figure 1 plots error as a function of the cleaned sample size on a corrupted TPCH dataset for a direct estimate, correction, and AllDirty (query on the full dirty data). In both cases, there is a break-even point (in terms of number of cleaned samples) when the data cleaning has mitigated more data error than the approximation error introduced by sampling. After this point, SampleClean improves query accuracy in comparison to AllDirty. When errors are relatively rare (5% corruption rate), the correction is more accurate. When errors are more significant (50% corruption rate), the direct estimate is more accurate. Note that the direct estimate returns results of the same accuracy regardless of the corruption rate.

3 SampleClean: Aggregate Query Processing on Dirty Data

This section introduces the SampleClean framework where the results of aggregate queries on dirty relations are estimated and bounded.

3.1 Problem Setup

Dirty Relation: Let R be a dirty relation corrupted with the following errors: (*Attribute Errors*) a row $r \in R$ has an attribute error in an attribute a if $r(a)$ is incorrect or has a missing value, (*Duplication Errors*) a row $r \in R$ is said to be a duplicate if there exists another distinct $r' \in R$ such that they refer to the same entity. For every dirty relation R, there is a cleaned version R_{clean} where attribute errors are corrected (or filled) and duplicates are merged to a canonical version.

Data Cleaning Model: For each row $r \in R$ the user-specified data cleaning technique $C(\cdot)$ must provide the following quantities: `Correct`(r) [a] the corrected value of the attribute, `Numdup`(r) the number of times the record is duplicated in the entire dataset.

Queries: SampleClean addresses aggregate queries of the form:

```
SELECT f(a) FROM R WHERE predicate GROUP BY gb_attrs
```

where f is `avg`, `sum`, or `count`.

SampleClean Problem Given a dirty relation R, and a user-specified data cleaning function $C(\cdot)$, the SampleClean problem is to estimate the result of an aggregate query q applied to the hypothetical cleaned relation $q(R_{clean})$ with a budget of applying C to at most k rows of R.

3.2 Sample Estimates

Consider a simpler problem; suppose we want to estimate the `mean` value of a set of real numbers R ignoring data error from a sample S. If S is sampled uniformly at random from R (with or without replacement), we can calculate the `mean` of S and for a large enough sample, the Central Limit Theorem (CLT) states that these estimates follow a normal distribution:

$$N(\texttt{mean}(R), \frac{var(R)}{k})$$

Since the estimate is normally distributed, we can define a confidence interval parametrized by λ (e.g., 95% indicates $\lambda = 1.96$)[2].

$$\texttt{mean}(S) \pm \lambda \sqrt{\frac{var(S)}{k}}. \tag{10}$$

It turns out that we can reformulate `sum`, `count`, and `avg` on an attribute a of a *relation* R as calculating a `mean` value so we can estimate their confidence intervals with the CLT $f(S) = \frac{1}{k}\sum_{r \in S} \phi(r)$. where $\phi(\cdot)$[3] expresses all of the necessary scaling to translate the query into a `mean` value calculation:

- `count`: $\phi(t) = \texttt{Predicate}(\text{r}) \cdot N$
- `sum`: $\phi(t) = \texttt{Predicate}(\text{r}) \cdot N \cdot r(a)$
- `avg`: $\phi(t) = \texttt{Predicate}(\text{r}) \cdot \frac{k}{k_{\texttt{pred}}} \cdot r(a)$

It turns out that these estimates are unbiased or conditionally unbiased; that is the expectation over all samples of the same size is the true answer.

[2]When estimating means from samples without replacement there is a finite population correction factor of $FPC = \frac{N-k}{N-1}$ which scales the confidence interval.

[3]`Predicate`(t) is the predicate of the aggregate query, where `Predicate`(t) = 1 or 0 denotes r satisfies or dissatisfies the predicate, respectively. $k_{\texttt{pred}}$ is the number of tuples that satisfy the predicate in the sample.

3.3 Direct Estimation with Data Errors

We are actually interested in estimating an aggregate query on $R_{\texttt{clean}}$. However, since we do not have the clean data, we cannot directly sample from $R_{\texttt{clean}}$. We must draw our sample from the dirty data R and then clean the sample. Running an aggregate query on the cleaned sample is not equivalent to computing the query result on a sample directly drawn from the clean data. Consider the case where data is duplicated, sampling from the dirty data leads to an over representation of the duplicated data in the sample. Even if cleaning is subsequently applied it does not change the fact that the sample is not uniform; and thus, the estimation method without errors presented before does not apply. Our goal is to define a new function $\phi_{\texttt{clean}}(\cdot)$, an analog to $\phi(\cdot)$, that corrects attribute values and re-scales to ensures that the estimate remains unbiased.

3.3.1 Attribute Errors

Attribute errors affect an individual row and thus do not change the sampling statistics. Consequently, if we apply the $\phi(\cdot)$ to the corrected tuple, we still preserve the uniform sampling properties of the sample S. In other words, the probability that a given tuple is sampled is not changed by the cleaning, thus we define $\phi_{\texttt{clean}}(t)$ as:

$$\phi_{\texttt{clean}}(t) = \phi\left(\texttt{Correct}(t)\right).$$

Note that the $\phi(\cdot)$ for an `avg` query is dependent on the parameter $k_{\texttt{pred}}$. If we correct values in the predicate attributes, we need to recompute $k_{\texttt{pred}}$ in the cleaned sample.

3.3.2 Duplication Errors

The duplicated data is more likely to be sampled and thus be over-represented in the estimate of the `mean`. We can address this with a weighted mean to reduce the effects of this over-representation. Furthermore, we can incorporate this weighting into $\phi_{\texttt{clean}}(\cdot)$. Specifically, if a tuple r is duplicated $m = \texttt{Numdup}(r)$ times, then it is m times more likely to be sampled, and we should down weight it with a $\frac{1}{m}$ factor compared to the other tuples in the sample. We formalize this intuition with the following lemma (proved in [45]):

Lemma 1: Let R be a population with duplicated tuples. Let $S \subseteq R$ be a uniform sample of size k. For each $r_i \in S$, let m_i denote its number of duplicates in R. (1) For `sum` and `count` queries, applying $\phi_{\texttt{clean}}(r_i) = \frac{\phi(r_i)}{m_i}$ yields an unbiased estimate; (2) For an `avg` query, the result has to be scaled by the duplication rate $d = \frac{k}{k'}$, where $k' = \sum_i \frac{1}{m_i}$, so using $\phi_{\texttt{clean}}(r_i) = d \cdot \frac{\phi(r_i)}{m_i}$ yields an unbiased estimate.

These results follow directly from importance sampling [32], where expected values can be estimated with respect to one probability measure, and corrected to reflect the expectation with respect to another.

3.3.3 Summary and Algorithm

In Table 1, we describe the transformation $\phi_{\texttt{clean}}(\cdot)$. Using this function, we formulate the direct estimation procedure:

1. Given a sample S and an aggregation function $f(\cdot)$
2. Apply $\phi_{\texttt{clean}}(\cdot)$ to each $t_i \in S$ and call the resulting set $\phi_{\texttt{clean}}(S)$
3. Calculate the mean μ_c, and the variance σ_c^2 of $\phi_{\texttt{clean}}(S)$
4. Return $\mu_c \pm \lambda\sqrt{\frac{\sigma_c^2}{K}}$

Table 1: $\phi_{\texttt{clean}}(\cdot)$ for `count`, `sum`, and `avg`. Note that N is the total size of dirty data (including duplicates).

Query	$\phi_{\texttt{clean}}(\cdot)$
`count`	$\texttt{Predicate(Correct(r))} \cdot N \cdot \frac{1}{\texttt{Numdup}(r)}$
`sum`	$\texttt{Predicate(Correct(r))} \cdot N \cdot \frac{\texttt{Correct}(r)[a]}{\texttt{Numdup}(r)}$
`avg`	$\texttt{Predicate(Correct(t))} \cdot \frac{dk}{k_{\texttt{pred}}} \cdot \frac{\texttt{Correct}(r)[a]}{\texttt{Numdup}(r)}$

3.4 Correction with Data Errors

Due to data errors, the result of the aggregation function f on the dirty population R differs from the true result $f(R) = f(R_{\texttt{clean}}) + \epsilon$. We derived a function $\phi_{\texttt{clean}}(\cdot)$ for the direct estimation. We contrasted this function with $\phi(\cdot)$ which does not clean the data. Therefore, we can write:

$$f(R) = \frac{1}{N}\sum_{r\in R}\phi(r) \qquad f(R_{\texttt{clean}}) = \frac{1}{N}\sum_{r\in R}\phi_{\texttt{clean}}(t)$$

If we solve for ϵ, we find that:

$$\epsilon = \frac{1}{N}\sum_{r\in R}\Big(\phi(r) - \phi_{\texttt{clean}}(r)\Big)$$

In other words, for every tuple r, we calculate how much $\phi_{\texttt{clean}}(r)$ changes $\phi(r)$. For a sample S, we can construct the set of differences between the two functions:

$$Q = \{\phi(r_1) - \phi_{\texttt{clean}}(r_1), \phi(r_2) - \phi_{\texttt{clean}}(r_2), \cdots, \phi(r_K) - \phi_{\texttt{clean}}(r_K)\}$$

The `mean` difference is an unbiased estimate of ϵ, the difference between $f(R)$ and $f(R_{\texttt{clean}})$. We can subtract this estimate from an existing aggregation of data to get an estimate of $f(R_{\texttt{clean}})$.

We derive the correction estimation procedure, which corrects an aggregation result:

1. Given a sample S and an aggregation function $f(\cdot)$
2. Apply $\phi(\cdot)$ and $\phi_{\texttt{clean}}(\cdot)$ to each $r_i \in S$ and call the set of differences $Q(S)$.
3. Calculate the mean μ_q, and the variance σ_q of $Q(S)$
4. Return $(f(R) - \mu_q) \pm \lambda\sqrt{\frac{\sigma_q^2}{k}}$

3.5 Analysis

Direct Estimate vs. Correction: In terms of the confidence intervals, we can analyze how direct estimation compares to correction for a fixed sample size k. Direct estimation gives an estimate that is proportional to the variance of the clean sample view: $\frac{\sigma_c^2}{k}$. Correction gives and estimate proportional to the variance of the *differences* before and after cleaning: $\frac{\sigma_q^2}{k}$. σ_q^2 can be rewritten as

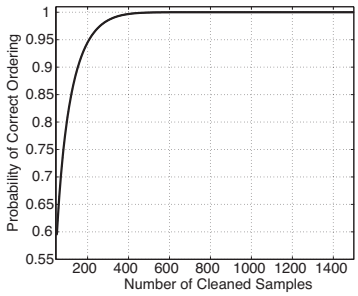
$$\sigma_c^2 + \sigma_q^2 - 2cov(S, S_{clean})$$

$cov(S, S_{clean})$ is the covariance between the the variables $\phi(r)$ and $\phi_{\texttt{clean}}(r)$. Therefore, a correction will have less variance when:

$$\sigma_S^2 \leq 2cov(S, S_{clean}) \tag{11}$$

If there are no errors $S_{clean} = S$ and then $cov(S, S_{clean}) = \sigma_c^2$ clearly satisfying the condition. Generally, if errors are small (i.e., the cleaned data is highly correlated with the dirty data) corrections will give higher accuracy. In practice, we can run both the correction and the direct estimate and take the one with a narrower confidence interval:

$$error^2 \leq O(\frac{\min\{\sigma_c^2, \sigma_q^2\}}{k}) \tag{12}$$



Name	Dirty	Clean	Pred %	Dup
Rakesh Agarwal	353	211	18.13%	1.28
Jeffery Ullman	460	255	05.00%	1.65
Michael Franklin	560	173	65.09%	1.13

Figure 2: We can return the correct ranking with 95% probability after cleaning only 210 total samples. To achieve a correct ranking with 99% probability, we require 326 samples to be cleaned.

Selectivity: Let p be the selectivity of the query and k be the sample size; that is, a fraction p records from the relation satisfy the predicate. For these queries, we can model selectivity as a reduction of effective sample size $k \cdot p$ making the estimate variance: $O(\frac{1}{k*p})$. Thus, the confidence interval's size is scaled up by $\frac{1}{\sqrt{p}}$. Just like there is a tradeoff between accuracy and maintenance cost, for a fixed accuracy, there is also a tradeoff between answering more selective queries and maintenance cost.

3.6 Results: Ranking Academic Authors

Microsoft maintains a public database of academic publications[4]. The errors in this dataset are primarily duplicated publications and mis-attributed publications. We selected publications from three database researchers: Jeffrey Ullman, Michael Franklin, and Rakesh Agarwal. To clean a sample of publications, we first manually removed the mis-attributions in the sample. Then, we applied the technique used in [44] to identify potential duplicates for all of publications in our sample, and manually examined the potential matches. For illustration purpose, we cleaned the entire dataset, and showed the cleaning results in Figure 2.

This table shows the difference between the reported number of publications (Dirty) and the number of publications after our cleaning (Clean). We also diagnosed the errors and recorded the duplication ratio (Dup) and the percentage of mis-attributed papers (Pred). Both Rakesh Agarwal and Michael Franklin had a large number of mis-attributed papers due to other authors with the same name (64 and 402 respectively). Jeffery Ullman had a comparatively larger number of duplicated papers (182).

If we were interested in ranking the authors, the dirty data would give us the wrong result. In Figure 2, we plot the probability of a correct ranking as a function of number of cleaned records with SampleClean. We show how we can return the correct ranking with 95% probability after cleaning only 210 total samples. To achieve a correct ranking with 99% probability, we require 326 samples to be cleaned. In comparison, AllDirty always returns an incorrect ranking. SampleClean provides a flexible way to achieve a desired confidence on decision based on dirty data queries.

4 View Cleaning: Stale Views are Dirty Data [30]

Suppose the relation R is in fact a derived relation V of an underlying dirty database D. We explored how we can efficiently apply a data cleaning operation to a sample of V. This extension has an important application in approximate Materialized View maintenance, where we model a stale Materialized View as dirty data, and the maintenance procedure as cleaning.

[4] http://academic.research.microsoft.com (Accessed Nov. 3, 2013)

4.1 Motivation

Some materialized views are computationally difficult to maintain and will have maintenance costs that can grow with the size of data (e.g, correlated aggregate in a sub-query). When faced with such challenges, it is common to batch updates to amortize maintenance overheads and add flexibility to scheduling. Like dirty data, any amount of staleness can lead to erroneous query results where the user has no idea about the magnitude or the scope of query error. Thus, we explore how samples of "clean" (up-to-date) data can be used for improved query processing on MVs without incurring the full cost of maintenance.

4.2 Notation and Definitions

View Cleaning returns a bounded approximation for aggregate queries on stale MVs for a flexible additional maintenance cost.

Materialized Views: Let $\mathcal{D}$ be a database which is a collection of relations $\{R_i\}$. A *materialized view* V is the result of applying a *view definition* to $\mathcal{D}$. View definitions are composed of standard relational algebra expressions: Select (σ_ϕ), Project (Π), Join ($\bowtie$), Aggregation (γ), Union ($\cup$), Intersection ($\cap$) and Difference ($-$).

Staleness: For each relation R_i there is a set of insertions ΔR_i (modeled as a relation) and a set of deletions ∇R_i. An "update" to R_i can be modeled as a deletion and then an insertion. We refer to the set of insertion and deletion relations as "delta relations", denoted by $\partial\mathcal{D}$:

$$\partial\mathcal{D} = \{\Delta R_1, ..., \Delta R_k\} \cup \{\nabla R_1, ..., \nabla R_k\}$$

A view S is considered *stale* when there exist insertions or deletions to any of its base relations. This means that at least one of the delta relations in $\partial\mathcal{D}$ is non-empty.

Maintenance: There may be multiple ways (e.g., incremental maintenance or re-computation) to maintain a view V, and we denote the up-to-date view as V'. We formalize the procedure to maintain the view as a *maintenance strategy* $\mathcal{M}$. A maintenance strategy is a relational expression the execution of which will return V'. It is a function of the database $\mathcal{D}$, the stale view V, and all the insertion and deletion relations $\partial\mathcal{D}$. In this work, we consider maintenance strategies composed of the same relational expressions as materialized views described above.

$$V' = \mathcal{M}(V, \mathcal{D}, \partial D)$$

Uniform Random Sampling: We define a sampling ratio $m \in [0, 1]$ and for each row in a view V, we include it into a sample with probability m. The relation S is a *uniform sample* of V if

$$(1)\ \forall s \in S : s \in V; \quad (2)\ Pr(s_1 \in S) = Pr(s_2 \in S) = m.$$

A sample is *clean* if and only if it is a uniform random sample of the up-to-date view V'.

4.3 Stale View Cleaning Problem

We are given a stale view S, a sample of this stale view S with ratio m, the maintenance strategy $\mathcal{M}$, the base relations $\mathcal{D}$, and the insertion and deletion relations $\partial\mathcal{D}$. We want to find a relational expression $\mathcal{C}$ such that:

$$V' = \mathcal{C}(S, \mathcal{D}, \partial\mathcal{D}),$$

where V' is a sample of the up-to-date view with ratio m.

Query Result Estimation: This problem can be addressed with the direct estimation and correction techniques described previously once we have a sample of up-to-date rows.

4.4 Cleaning a Sample View

We need to find an efficient maintnenance plan that avoids extra effort (i.e., materialization of rows outside the sample). The challenge is that $\mathcal{M}$ does not always commute with sampling. To address the commutativity problem, we need to ensure that for each $s \in V'$ all contributing rows in subexpressions to s are also sampled. We address this with a two-step process: (1) build a relation expression tree that preserves primary key relationships, and (2) use a hashing operator to push down along these relationships.

Primary Key: We recursively define a set of primary keys for all relations in the expression tree to define tuple provenance. The primary keys allow us to determine the set of rows that contribute to a row r in a derived relation. The following rules define a constructive definition for these keys:

Definition 2 (Primary Key Generation): For every relational expression R, we define the primary key attribute(s) of every expression to be:

- Base Case: All relations (leaves) must have an attribute p which is designated as a primary key.
- $\sigma_\phi(R)$: Primary key of the result is the primary key of R
- $\Pi_{(a_1,\ldots,a_k)}(R)$: Primary key of the result is the primary key of R. The primary key must always be included in the projection.
- $\bowtie_{\phi(r1,r2)}(R_1, R_2)$: Primary key of the result is the tuple of the primary keys of R_1 and R_2.
- $\gamma_{f,A}(R)$: The primary key of the result is the group by key A (which may be a set of attributes).
- $R_1 \cup R_2$: Primary key of the result is the union of the primary keys of R_1 and R_2
- $R_1 \cap R_2$: Primary key of the result is the intersection of the primary keys of R_1 and R_2
- $R_1 - R_2$: Primary key of the result is the primary key of R_1

For every node at the expression tree, these keys are guaranteed to uniquely identify a row.

Hashing: Next, instead of sampling with pseudo-random number generation, we use a hashing procedure. This procedure is a deterministic way of mapping a primary key to a Boolean, we can ensure that all contributing rows are also sampled. Let us denote the hashing operator $\eta_{a,m}(R)$. For all tuples in R, this operator applies a hash function whose range is $[0, 1]$ to primary key a (which may be a set) and selects those records with hash less than or equal to m. In a process analgous to predicate push down, we can optimize the maintenance expression by applying $\eta_{a,m}(\mathcal{M})$. The result $\mathcal{C}$ is an optimized maintenance plan expression that materializes a sample of rows.

Definition 3 (Hash push-down): For a derived relation R, the following rules can be applied to push $\eta_{a,m}(R)$ down the expression tree.

- $\sigma_\phi(R)$: Push η through the expression.
- $\Pi_{(a_1,\ldots,a_k)}(R)$: Push η through if a is in the projection.
- $\bowtie_{\phi(r1,r2)}(R_1, R_2)$: Push down is possible for foreign key equality joins.
- $\gamma_{f,A}(R)$: Push η through if a is in the group by clause A.
- $R_1 \cup R_2$: Push η through to both R_1 and R_2
- $R_1 \cap R_2$: Push η through to both R_1 and R_2
- $R_1 - R_2$: Push η through to both R_1 and R_2

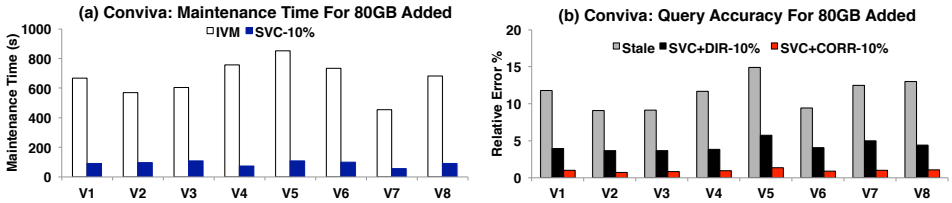


Figure 3: (a) We compare the maintenance time of View Cleaning with a 10% sample and full incremental maintenance (IVM). (b) We also evaluate the accuracy of the estimation techniques: (Direct DIR), Correction (CORR), and Dirty (Stale).

4.5 Results: Video Streaming Log Analysis

We evaluate View Cleaning on Apache Spark 1.1.0 with 1TB of logs from a video streaming company, Conviva [2]. This is a denormalized user activity log corresponding to video views and various metrics such as data transfer rates, and latencies. Accompanying this data is a four month trace of queries in SQL. We identified 8 common summary statistics-type queries that calculated engagement and error-diagnosis metrics. We populated these view definitions using the first 800GB of user activity log records. We then applied the remaining 200GB of user activity log records as the updates (i.e., in the order they arrived) in our experiments. We generated aggregate random queries over this view by taking either random time ranges or random subsets of customers.

In Figure 3(a), we show that on average over all the views, View Cleaning with a 10% sample gives a 7.5x speedup. For one of the views full incremental maintenance takes nearly 800 seconds, even on a 10-node cluster, which is a very significant cost. In Figure 3(b), we show that View Cleaning also gives highly accurate results with an average error of 0.98% for the correction estimate. This experiment highlights a few salient benefits of View Cleaning: (1) sampling is a relatively cheap operation and the relative speedups in a single node and distributed environment are similar, (2) for analytic workloads like Conviva (i.e., user engagement analysis) a 10% sample gives results with 99% accuracy, and (3) savings are still significant in systems like Spark that do not support selective updates.

5 ActiveClean: Machine Learning on Dirty Data [31]

Analytics is moving beyond SQL, and the growing popularity of predictive models [1, 7, 17, 28] leads to additional challenges in managing dirty data.

5.1 Simpson's Paradox

The challenge is that the high-dimensional models are very sensitive to systematic biases, and many of the techniques applied in practice suffer methodological problems. Consider the following approach: let k rows be cleaned, but all of the remaining dirty rows are retained in the dataset. Figure 4 highlights the dangers of this approach on a very simple dirty dataset and a linear regression model i.e., the best fit line for two variables. One of the variables is systematically corrupted with a translation in the x-axis (Figure 4a). The dirty data is marked in brown and the clean data in green, and their respective best fit lines are in blue. After cleaning only two of the data points (Figure 4b), the resulting best fit line is in the opposite direction of the true model. This is a well-known phenomenon called Simpsons paradox, where mixtures of different populations of data can result in spurious relationships [41]. Training models on a mixture of dirty and clean data can lead to unreliable results, where artificial trends introduced by the mixture can be confused for the effects of data cleaning. Figure 4c also illustrates that, even in two dimensions, models trained from small samples can be as incorrect as the mixing solution described before.

Figure 4: (a) Systematic corruption in one variable can lead to a shifted model. (b) Mixed dirty and clean data results in a less accurate model than no cleaning. (c) Small samples of only clean data can result in similarly inaccurate models.

5.2 Problem Setup

This work focuses on a class of well analyzed predictive analytics problems; ones that can be expressed as the minimization of convex loss functions. Examples includes all generalized linear models (including linear and logistic regression), all variants of support vector machines, and in fact, `avg` and `median` are also special cases.

Formally, for labeled training examples $\{(x_i, y_i)\}_{i=1}^N$, the problem is to find a vector of *model parameters* θ by minimizing a loss function ϕ over all training examples:

$$\theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \phi(x_i, y_i, \theta)$$

Where ϕ is a convex function in θ. Without loss of generality, we will include regularization as part of the loss function i.e., $\phi(x_i, y_i, \theta)$ includes $r(\theta)$.

Definition 4 (Convex Data Analytics): A convex data analytics problem is specified by a set of features X, corresponding set of labels Y, and a parametrized loss function ϕ that is convex in its parameter θ. The result is a **model** θ that minimizes the sum of losses over all features and labels.

ActiveClean Problem: Let R be a dirty relation, $F(r) \mapsto (x, y)$ be a featurization that maps a record $r \in R$ to a feature vector x and label y, ϕ be a convex regularized loss, and $C(r) \mapsto r_{clean}$ be a cleaning technique that maps a record to its cleaned value. Given these inputs, the ActiveClean problem is to return a **reliable** estimate $\hat{\theta}$ of the clean model for any limit k on the number of times the data cleaning $C(\cdot)$ can be applied.

Reliable precisely means that the expected error in this estimate (i.e., L2 difference w.r.t a model trained on a fully cleaned dataset) is bounded above by a monotonically decreasing function in k and a monotonically decreasing function of the error of the dirty model. In other words, more cleaning implies more accuracy, and less initial error implies faster convergence.

5.3 Model Updates

The main insight of this work is that, in Convex Data Analytics, sampling is naturally part of the query processing. Mini-batch stochastic gradient descent (SGD) is an algorithm for finding the optimal value given the convex loss and data. In mini-batch SGD, random subsets of data are selected at each iteration and the average gradient is computed for every batch. Instead of calculating the average gradient for the batch w.r.t to the dirty data, we apply data cleaning at that point–inheriting the convergence bounds from batch SGD. It is well known that even for an arbitrary initialization SGD makes significant progress in less than one epoch (a pass through the entire dataset) [10]. Furthermore in this setting, the dirty model can be much more accurate than an arbitrary initialization; leading to highly accurate models without processing the entire data.

ActiveClean is initialized with $\theta^{(1)} = \theta^{(d)}$ which is the dirty model. At each iteration $t = \{1, ..., T\}$, the cleaning is applied to a batch of data b selected from the set of candidate dirty rows R. Then, an average gradient is estimated from the cleaned batch and the model is updated. Iterations continue until $k = T \cdot b$ rows are cleaned. The user sets the learning rate λ initially.

1. Calculate the gradient over the sample of clean data and call the result $g_S(\theta^{(t)})$
2. Apply the following update rule:
$$\theta^{(t+1)} \leftarrow \theta^{(t)} - \lambda \cdot g_S(\theta^{(t)})$$

5.4 Optimizations

ActiveClean has a number of additional optimizations exploiting the structure of Convex Data Analytics problems.

Detector: In this step, the detector select a candidate set of dirty rows $R_{dirty} \subseteq R$. There are two techniques to do this: (1) an *a priori* case, and (2) and an adaptive case. In the *a priori* case, the detector knows which data is dirty in advance. In the adaptive case, the detector learns classifier based on previously cleaned data to detect corruption in uncleaned data. This allows ActiveClean to prioritize cleaning data expected to be dirty.

Non-uniform Sampler: The sampler draws a sample of rows $S_{dirty} \subseteq R_{dirty}$. This is a non-uniform sample where each record r has a sampling probability $p(r)$. We derive the theoretical minimum variance sampling distribution, which is impossible to realize as it requires knowing the clean data in advance. Therefore, we use a first order first-order approximation of this distribution based on estimates of the clean data.

Estimator: The estimator approximates the optimal distribution derived in the Sample step. Based on the change in the featurized data $F(S_{clean})$ and $F(S_{dirty})$, it directs the next iteration of sampling to select points that will have changes most valuable to the next model update.

6 Related Work

Approximate Query Processing: AQP has been studied for more than two decades [23, 16]. Many AQP approaches [12, 5, 40, 8, 27, 37, 15, 47] were proposed, aiming to enable interactive query response times. There are also many studies on creating other synopsis of the data, such as histograms or wavelets [16]. While a substantial works on approximate query processing, these works mainly focus on how to deal with sampling errors, with little attention to data errors.

Data Cleaning: There have been many studies on various data-cleaning techniques, such as rule-based approaches [21, 18], outlier detection [26, 19], filling missing values, and duplicate detection [13, 9, 44]. In order to ensure reliable cleaning results, most of these techniques require human involvement. For example, Fan et al. [22] proposed to employ editing rules, master data and user confirmation to clean data, and proved that their approaches can always lead to correct cleaning results. Wang et al. [44] proposed a hybrid human-machine approach to detect duplicate entities in data, which can achieve higher detection accuracy than machine-only approaches. In SampleClean, the main focus is not on a specific data-cleaning technique, but rather on a new framework that enables a flexible trade-off between data cleaning cost and result quality. Indeed, we can apply any data-cleaning technique to clean the sample data, and then utilize our framework to estimate query results based on the cleaned sample.

Views and Cleaning: Meliou et al. [33] proposed a technique to trace errors in an MV to base data and find responsible erroneous tuples. They do not, however, propose a technique to correct the errors as in View Cleaning. Correcting general errors as in Meliou et al. is a hard constraint satisfaction problem. However,

in View Cleaning, through our formalization of staleness, we have a model of how updates to the base data (modeled as errors) affect MVs, which allows us to both trace errors and clean them. Wu and Madden [46] did propose a model to correct "outliers" in an MV through deletion of records in the base data. This is a more restricted model of data cleaning than View Cleaning, where the authors only consider changes to existing rows in an MV (no insertion or deletion) and do not handle the same generality of relational expressions (e.g., nested aggregates). Challamalla et al. [11] proposed an approximate technique for specifying errors as constraints on a materialized view and proposing changes to the base data such that these constraints can be satisfied. While complementary, one major difference between the three works [33, 46, 11] and View Cleaning is that they require an explicit specification of erroneous rows in a materialized view. Identifying whether a row is erroneous requires materialization and thus specifying the errors is equivalent to full incremental maintenance. However, while these approaches are not directly applicable for staleness, we see View Cleaning as complementary to these works in the dirty data setting.

7 Future Work and Open Problems

We further describe a number of open theoretical and practical problems to challenge the community:

Sample-based Optimization of Workflows: In practical data cleaning workflows, there are numerous design choices e.g., whether or not to use crowdsourcing, similarity functions, etc. An open problem is using samples of cleaned data to estimate and tune parameters on data cleaning workflows.

Optimality: For aggregate queries in the budgeted data cleaning setting, variance of the clean data σ_c^2, variance of the pairwise differences between clean and dirty data σ_d^2, and sample size k, is $O(\frac{\min\{\sigma_c,\sigma_d\}}{\sqrt{k}})$ (derived in this work) an optimal error bound? By optimal error bound, we mean that given no other information about the data distribution, the bound cannot be tightened.

Point-Lookup Dichotomy: This work focuses on aggregate analytics such as queries and statistical models. In fact, as the selectivity of the analytics goes to 0 (i.e., single row lookup), the bounds in this work limit to infinity. However, in practice, cleaning a sample of data can be used to address such queries, where a statistical model can be trained on a sample of data to learn a mapping between dirty and clean data. An open problem is exploring how much looser is a generalization bound (e.g., via Learning Theory) compared to the bounds on aggregate queries.

Confirmation Bias and Sample Re-use: Confirmation bias is defined as a "tendency to search for or interpret information in a way that confirms one's preconceptions"[39]. In systems like SampleClean, users repeatedly query and clean the sample of data. This process may encourage confirmation bias as users are allowed to modify data based on reviewing a query result (i.e., what prevents a user from removing data that does not match his or her hypothesis). An open problem is designing efficient APIs to mitigate the effects of *confirmation bias*, perhaps by limiting the number of times a user can query the sample to review the effects of a cleaning operation.

8 Conclusion

An important challenge in data analytics is presence of dirty data in the form of missing, duplicate, incorrect or inconsistent values. Data analysts report that data cleaning remains one of the most time consuming steps in the analysis process, and data cleaning can require a significant amount of developer effort in writing software or rules to fix the corruption. SampleClean studies the integration of Sample-based Approximate Query Processing and data cleaning; to provide analysts a tradeoff between cleaning the entire dataset and avoiding cleaning altogether. To the best of our knowledge, this is the first work to marry data cleaning with sampling-based query processing. While sampling introduces approximation error, the data cleaning mitigates errors in query

results. This idea opened up a number of new research opportunities, and we applied the same principles to other domains such as Materialized View Maintenance and Machine Learning.

We would like to thank Mark Wegman whose ideas helped inspire SampleClean project. This research would not have been possible without collaboration with Daniel Haas and Juan Sanchez. We would also like to acknowledge Kai Zeng, Ben Recht, and Animesh Garg for their input, feedback, and advice throughout the course of this research. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

References

[1] Berkeley data analytics stack. `https://amplab.cs.berkeley.edu/software/`.

[2] Conviva. `http://www.conviva.com/`.

[3] For big-data scientists, 'janitor work' is key hurdle to insights. http://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html.

[4] Sampleclean. http://sampleclean.org/, 2015.

[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The aqua approximate query answering system. In *SIGMOD Conference*, pages 574–576, 1999.

[6] 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.

[7] A. Alexandrov, R. Bergmann, S. Ewen, J. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The stratosphere platform for big data analytics. *VLDB J.*, 23(6):939–964, 2014.

[8] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD Conference*, pages 539–550, 2003.

[9] M. Bilenko and R. J. Mooney. Adaptive duplicate detection using learnable string similarity measures. In *KDD*, pages 39–48, 2003.

[10] L. Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[11] A. Chalamalla, I. F. Ilyas, M. Ouzzani, and P. Papotti. Descriptive and prescriptive data cleaning. In *SIGMOD Conference*, pages 445–456, 2014.

[12] S. Chaudhuri, G. Das, and V. R. Narasayya. Optimized stratified sampling for approximate query processing. *ACM Trans. Database Syst.*, 32(2):9, 2007.

[13] P. Christen. Febrl: a freely available record linkage system with a graphical user interface. In *HDKM*, pages 17–25, 2008.

[14] X. Chu, J. Morcos, I. F. Ilyas, M. Ouzzani, P. Papotti, N. Tang, and Y. Ye. KATARA: A data cleaning system powered by knowledge bases and crowdsourcing. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data, Melbourne, Victoria, Australia, May 31 - June 4, 2015*, pages 1247–1261, 2015.

[15] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, J. Gerth, J. Talbot, K. Elmeleegy, and R. Sears. Online aggregation and continuous query support in mapreduce. In *SIGMOD Conference*, pages 1115–1118, 2010.

[16] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[17] A. Crotty, A. Galakatos, and T. Kraska. Tupleware: Distributed machine learning on small clusters. *IEEE Data Eng. Bull.*, 37(3):63–76, 2014.

[18] M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. F. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD Conference*, pages 541–552, 2013.

[19] T. Dasu and T. Johnson. *Exploratory data mining and data cleaning*. Wiley, 2003.

[20] X. L. Dong and D. Srivastava. Big data integration. *PVLDB*, 6(11):1188–1189, 2013.

[21] W. Fan and F. Geerts. Foundations of data quality management. *Synthesis Lectures on Data Management*, 2012.

[22] W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *PVLDB*, 3(1):173–184, 2010.

[23] M. N. Garofalakis and P. B. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001.

[24] C. Gokhale, S. Das, A. Doan, J. F. Naughton, N. Rampalli, J. Shavlik, and X. Zhu. Corleone: Hands-off crowdsourcing for entity matching. In *SIGMOD*, 2014.

[25] D. Haas, S. Krishnan, J. Wang, M. J. Franklin, and E. Wu. Wisteria: Nurturing scalable data cleaning infrastructure. *Proceedings of the VLDB Endowment*, 8(12), 2015.

[26] J. M. Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

[27] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD Conference*, pages 171–182, 1997.

[28] J. M. Hellerstein, C. Ré, F. Schoppmann, D. Z. Wang, E. Fratkin, A. Gorajek, K. S. Ng, C. Welton, X. Feng, K. Li, and A. Kumar. The madlib analytics library or MAD skills, the SQL. *PVLDB*, 5(12):1700–1711, 2012.

[29] S. R. Jeffery, M. J. Franklin, and A. Y. Halevy. Pay-as-you-go user feedback for dataspace systems. In *SIGMOD Conference*, pages 847–860, 2008.

[30] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and T. Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *Proceedings of the VLDB Endowment*, 8(12), 2015.

[31] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, and E. Wu. Activeclean: Progressive data cleaning for convex data analytics. 2015.

[32] J. S. Liu. Metropolized independent sampling with comparisons to rejection sampling and importance sampling. *Statistics and Computing*, 6(2):113–119, 1996.

[33] A. Meliou, W. Gatterbauer, S. Nath, and D. Suciu. Tracing data errors with view-conditioned causality. In *SIGMOD Conference*, pages 505–516, 2011.

[34] F. Olken. *Random sampling from databases*. PhD thesis, University of California, 1993.

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

[36] F. Olken and D. Rotem. Maintenance of materialized views of sampling queries. In *ICDE*, pages 632–641, 1992.

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

[38] H. Park and J. Widom. Crowdfill: collecting structured data from the crowd. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, pages 577–588, 2014.

[39] S. Plous. *The psychology of judgment and decision making.* Mcgraw-Hill Book Company, 1993.

[40] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, pages 296–301, 2011.

[41] E. H. Simpson. The interpretation of interaction in contingency tables. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 238–241, 1951.

[42] N. Swartz. Gartner warns firms of 'dirty data'. *Information Management Journal*, 41(3), 2007.

[43] J. R. Taylor. An introduction to error analysis: The study of uncertainties in physical measurements, 327 pp. *Univ. Sci. Books, Mill Valley, Calif*, 1982.

[44] J. Wang, T. Kraska, M. J. Franklin, and J. Feng. Crowder: Crowdsourcing entity resolution. *PVLDB*, 5(11):1483–1494, 2012.

[45] J. Wang, S. Krishnan, M. J. Franklin, K. Goldberg, T. Kraska, and T. Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *SIGMOD Conference*, pages 469–480, 2014.

[46] E. Wu and S. Madden. Scorpion: Explaining away outliers in aggregate queries. *PVLDB*, 6(8):553–564, 2013.

[47] S. Wu, S. Jiang, B. C. Ooi, and K.-L. Tan. Distributed online aggregation. *PVLDB*, 2(1):443–454, 2009.

[48] M. Yakout, A. K. Elmagarmid, J. Neville, M. Ouzzani, and I. F. Ilyas. Guided data repair. *PVLDB*, 2011.

Independent Range Sampling on a RAM

Xiaocheng Hu Miao Qiao Yufei Tao

Department of Computer Science and Engineering
Chinese University of Hong Kong
New Territories, Hong Kong
{xchu, mqiao, taoyf}@cse.cuhk.edu.hk

Abstract

This invited paper summarizes our results on the "independent range sampling" problem in the RAM computation model. The input is a set P of n points in $\mathbb{R}$. Given an interval $q = [x, y]$ and an integer $t \geq 1$, a query returns t elements uniformly sampled (with/without replacement) from $P \cap q$. The sampling result must be independent from those returned by the previous queries. The objective is to store P in a structure for answering all queries efficiently. If no updates are allowed, there is a trivial $O(n)$-size structure that guarantees $O(\log n + t)$ query time. The problem is much more interesting when P is dynamic (allowing both insertions and deletions). The state of the art is a structure of $O(n)$ space that answers a query in $O(t \log n)$ time, and supports an update in $O(\log n)$ time. We describe a new structure of $O(n)$ space that answers a query in $O(\log n + t)$ expected time, and supports an update in $O(\log n)$ time.

1 Introduction

A *reporting* query, in general, retrieves from a dataset all the elements satisfying a condition. In the current big data era, such a query easily turns into a "big query", namely, one whose result contains a huge number of elements. In this case, even the simple task of enumerating all those elements can be extremely time consuming. This phenomenon naturally brings back the notion of *query sampling*, a classic concept that was introduced to the database community several decades ago. The goal of query sampling is to return, instead of an entire query result, only a random sample set of the elements therein. The usefulness of such a sample set has long been recognized even in the non-big-data days (see an excellent survey in [12]). The unprecedented gigantic data volume we are facing nowadays has only strengthened the importance of query sampling. Particularly, this is an effective technique in dealing with the big-query issue mentioned earlier in many scenarios where acquiring a query result in its entirety is not compulsory.

This work aims to endow query sampling with *independence*; namely, the samples returned by each query should be independent from the samples returned by the previous queries. In particular, we investigate how to achieve this purpose on *range reporting*, as it is a very fundamental query in the database and data structure fields. Formally, the problem we study can be stated as follows:

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Problem 1 (Independent Range Sampling (IRS)): Let P be a set of n points in $\mathbb{R}$. Given an interval $q = [x, y]$ in $\mathbb{R}$ and an integer $t \geq 1$, we define two types of queries:

- A **with replacement (WR) query** returns a sequence of t points, each of which is taken uniformly at random from $P(q) = P \cap q$.
- Requiring $t \leq |P(q)|$, a **without replacement (WoR) query** returns a subset R of $P(q)$ with $|R| = t$, which is taken uniformly at random from all the size-t subsets of $P(q)$. The query may output the elements of R in an arbitrary order.

In both cases, the output of the query must be independent from the outputs of all previous queries. □

Guaranteeing independence among the sampling results of all queries ensures a strong sense of fairness: the elements satisfying a query predicate always have the same chance of being reported (regardless of the samples returned previously), as is a desirable feature in battling the "big-query issue". Furthermore, the independence requirement also offers convenience in statistical analysis and algorithm design. In particular, it allows one to issue the *same* query multiple times to fetch different samples. This is especially useful when one attempts to test a property by sampling, but is willing to accept only a small failure probability of drawing a wrong conclusion. The independence guarantees that the failure probability decreases exponentially with the number of times the query is repeated.

Computation Model. We study IRS on a *random access machine* (RAM), where it takes constant time to perform a comparison, a $+$ operation, and to access a memory location. For randomized algorithms, we make the standard assumption that it takes constant time to generate a random integer in $[0, 2^w - 1]$, where w is the length of a word.

Existing Results. Next we review the literature on IRS, assuming the WR semantics—we will see later that a WoR query with parameters q, t can be answered by a constant number (in expectation) of WR queries having parameters $q, 2t$.

The problem is trivial when P is static. Specifically, we can simply store the points of P in ascending order using an array A. Given a query with parameters $q = [x, y]$ and t, we can first perform binary search to identify the subsequence in A that consists of the elements covered by q. Then, we can simply sample from the subsequence by generating t random ranks and accessing t elements. The total query cost is $O(\log n + t)$.

The problem becomes much more interesting when P is dynamic, namely, it admits insertions and deletions of elements. This problem was first studied more than two decades ago. The best solution to this date uses $O(n)$ space, answers a query in $O(t \log n)$ time, and supports an update in $O(\log n)$ time (see [12] and the references therein). This can be achieved by creating a "rank structure" on P that allows us to fetch the i-th (for any $i \in [1, n]$) largest element of P in $O(\log n)$ time. After this, we can then simulate the static algorithm described earlier by spending $O(\log n)$ time, instead of $O(1)$, fetching each sample.

If one does not require independence of the sampling results of different queries, query sampling can be supported as follows. For each $i = 0, 1, ..., \lceil \log n \rceil$, maintain a set P_i by independently including each element of P with probability $1/2^i$. Given a query with interval $q = [x, y]$, $P_i \cap q$ serves as a sample set where each element in $P(q)$ is taken with probability $1/2^i$. However, by issuing the same query again, one always gets back the same samples, thus losing the benefits of IRS mentioned before.

Also somewhat relevant is the recent work of Wei and Yi [16], in which they studied how to return various statistical summaries (e.g., quantiles) on the result of range reporting. They did not address the problem of query sampling, let alone how to enforce the independence requirement. At a high level, IRS may be loosely classified as a form of *online aggregation* [8], because most research on this topic has been devoted to the maintenance of a random sample set of a long-running query (typically, aggregation from a series of joins); see [10] and the

references therein. As far as IRS is concerned, we are not aware of any work along this line that guarantees better performance than the solutions surveyed previously.

It is worth mentioning that sampling algorithms have been studied extensively in various contexts (for entry points into the literature, see [1, 3, 4, 6, 7, 11, 14, 15]). These algorithms aim at efficiently producing sample sets for different purposes over a static or evolving dataset. Our focus, on the other hand, is to design data structures for sampling the results of arbitrary range queries.

Our Results. Recall that a query specifies two parameters: a range $q = [x, y]$ and the number t of samples. We say that the query is *one-sided* if $x = -\infty$ or $y = \infty$; otherwise, the query is *two-sided*. We will describe a dynamic structure of $O(n)$ space that answers a two-sided WR or WoR query in $O(\log n + t)$ expected time, and supports an update in $O(\log n)$ time (all the expectations in this paper depend only on the random choices made by our algorithms). For one-sided queries, the query time can be improved to $O(\log \log n + t)$ expected, while retaining the same space and update complexities. Besides their excellent theoretical guarantees, all of our structures have the additional advantage of being fairly easy to implement.

Assuming the WR semantics, we will first describe a structure for one-sided queries (Section 2), before attending to two-sided ones (Sections 3 and 4). In Section 5, we will explain how to answer WoR queries.

2 A One-Sided Structure

Structure. We build a *weight-balanced B-tree* (WBB-tree) [2] on the input set P with leaf capacity $b = 4$ and branching parameter $f = 8$. In general, a WBB-tree parameterized by b and f is a B-tree where

- data elements are stored in the leaves. We label the leaf level as level 0; if a node is at level i, then its parent is at level $i + 1$.
- a non-root node u at the i-th level has between $bf^i/4$ and bf^i elements stored in its subtree. We denote by $P(u)$ the set of those elements. This property implies that an internal node has between $f/4$ and $4f$ child nodes.

Each node u is naturally associated with an interval $I(u)$ defined as follows. If u is a leaf, then $I(u) = (e', e]$ where e (or e', resp.) is the largest element stored in u (or the leaf preceding u, resp.); specially, if no leaf precedes u, then $e' = -\infty$. If u is an internal node, then $I(u)$ unions the intervals of all the child nodes of u.

Let z_ℓ be the leftmost leaf (i.e., the leaf containing the smallest element of P). Denote by Π_ℓ the path from the root to z_ℓ. For every node u on Π_ℓ, store all the elements of $P(u)$ in an array $A(u)$. Note that the element ordering in $A(u)$ is arbitrary. The total space of all arrays is $O(n)$, noticing that the arrays' sizes shrink geometrically as we descend Π_ℓ.

Query. A one-sided query with parameters $q = (-\infty, y]$ and t is answered as follows. We first identify the lowest node u on Π_ℓ such that $I(u)$ fully covers q. If u is a leaf, we obtain the entire $P(q) = P \cap q$ from u in constant time, after which the samples can be obtained trivially in $O(t)$ time. If u is an internal node, we obtain a sequence $\mathcal{R}$ by repeating the next step until the length of $\mathcal{R}$ is t: select uniformly at random an element e from $A(u)$, and append e to $\mathcal{R}$ if e is covered by q. We return $\mathcal{R}$ as the query's output. Note that the $\mathcal{R}$ computed this way is independent from all the past queries.

We argue that the above algorithm runs in $O(\log \log n + t)$ expected time, focusing on the case where u is not a leaf. Let $k = |P(q)|$. Node u can be found in $O(\log \log n)$ time by creating a binary search tree on the intervals of the nodes on Π_ℓ. It is easy to see that the size of $A(u)$ is at least k but at most ck for some constant $c \geq 1$. Hence, a random sample e from $A(u)$ has at least $1/c$ probability of falling in q. This implies that we expect to sample no more than $ct = O(t)$ times before filling up $\mathcal{R}$.

Update. Recall the well-known fact that an array can be maintained in $O(1)$ time per insertion and deletion[1]—this is true even if the array's size needs to grow or shrink—provided that the element ordering in the array does not matter. The key to updating our structure lies in modifying the secondary arrays along Π_ℓ. Whenever we insert/delete an element e in the subtree of a node u on Π_ℓ, e must be inserted/deleted in $A(u)$ as well. Insertion is easy: simply append e to $A(u)$. To delete e, we first locate e in $A(u)$, swap it with the last element of $A(u)$, and then shrink the size of $A(u)$ by 1. The problem, however, is how to find the location of e; although hashing does this trivially, the update time becomes $O(\log n)$ expected.

The update time can be made worst case by slightly augmenting our structure. For each element $e \in P$, we maintain a linked list of all its positions in the secondary arrays. This linked list is updated in constant time whenever a position changes (this requires some proper bookkeeping, e.g., pointers between a position in an array and its record in a linked list). In this way, when e is deleted, we can find all its array positions in $O(\log n)$ time. Taking care of other standard details of node balancing (see [2]), we have arrived at:

Theorem 1: For the IRS problem, there is a RAM structure of $O(n)$ space that can answer a one-sided WR query in $O(\log \log n + t)$ expected time, and can be updated in $O(\log n)$ worst-case time per insertion and deletion.

3 A 2-Sided Structure of $O(n \log n)$ Space

By applying standard range-tree ideas to the one-sided structure in Theorem 1, we obtain a structure for two-sided queries with space $O(n \log n)$ and query time $O(\log n + t)$ expected. However, it takes $O(\log^2 n)$ time to update the structure. Next, we give an alternative structure with improved update cost.

Structure. Again, we build a WBB-tree T on the input set P with leaf capacity $b = 4$ and branching parameter $f = 8$. At each node u in the tree, we keep a count equal to $|P(u)|$, i.e., the number of elements in its subtree. We also associate u with an array $A(u)$ that stores all the elements of $P(u)$; the ordering in $A(u)$ does not matter. The overall space consumption is clearly $O(n \log n)$.

Query. We will see how to use the structure to answer a query with parameters $q = [x, y]$ and t. Let $k = |P(q)|$. Since we aim at query time of $\Omega(\log n)$, it suffices to consider only $k > 0$ (one can check whether $k > 0$ easily with a separate "range count" structure). The crucial step is to find at most two nodes u_1, u_2 satisfying two conditions:

- $\boldsymbol{c1}$ $I(u_1)$ and $I(u_2)$ are disjoint, and their union covers q;
- $\boldsymbol{c2}$ $|P(u_1)| + |P(u_2)| = O(k)$.

These nodes can be found as follows. First, identify the lowest node u in T such that $I(u)$ covers q. If u is a leaf node, setting $u_1 = u$ and $u_2 = nil$ satisfies both conditions.

Now, suppose that u is an internal node. If q spans the interval $I(u')$ of at least one child u' of u, then once again setting $u_1 = u$ and $u_2 = nil$ satisfies both conditions. Now, consider that q does not span the interval of any child of u. In this case, x and y must fall in the intervals of two consecutive child nodes u', u'' of u, respectively. Define $q_1 = q \cap I(u')$ and $q_2 = q \cap I(u'')$. We decide u_1 (u_2, resp.) as the lowest node in the subtree of u' (u'', resp.) whose interval covers q_1 (q_2, resp.); see Figure 1 for an illustration. The lemma below shows that our choice is correct.

Lemma 2: The u_1 and u_2 we decided satisfy conditions $\boldsymbol{c1}$ and $\boldsymbol{c2}$.

[1] A deletion needs to specify where the target element is in the array.

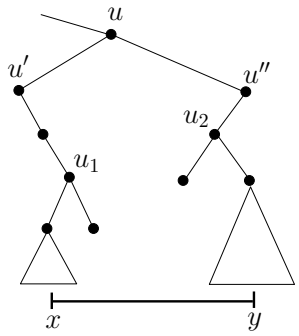


Figure 1: Answering a query at two nodes

Proof: We will focus on the scenario where u is an internal node. Let k_1 (k_2, resp.) be the number of elements in the subtree of u' (u'', resp.) covered by q. Clearly, $k = k_1 + k_2$. It suffices to show that $|P(u_1)| = O(k_1)$ and $|P(u_2)| = O(k_2)$. We will prove only the former due to symmetry. In fact, if u_1 is a leaf, then both k_1 and $|P(u_1)|$ are $O(1)$. Otherwise, q definitely spans the interval of a child node, say $\hat{u}$, of u_1. Hence, $|P(u_1)| = O(|P(\hat{u})|) = O(k_1)$. □

Let us continue the description of the query algorithm, given that u_1 and u_2 are already found. We conceptually append $A(u_1)$ to $A(u_2)$ to obtain a concatenated array A. Then, we repetitively perform the following step until an initially empty sequence $\mathcal{R}$ has length t: sample uniformly at random an element e from A, and append e to $\mathcal{R}$ if it lies in q. Note that since we know both $|A(u_1)|$ and $|A(u_2)|$, each sample can be obtained in constant time. Since A has size $O(k)$ and at least k elements covered by q, we expect to sample $O(t)$ elements before filling up $\mathcal{R}$. The total query cost is therefore $O(\log n + t)$ expected.

Update. The key to updating our structure is to modify the secondary arrays, as can be done using the ideas explained in Section 2 for updating our one-sided structure. The overall update time is $O(\log n)$.

Lemma 3: For the IRS problem, there is a RAM structure of $O(n \log n)$ space that can answer a two-sided WR query in $O(\log n + t)$ expected time, and can be updated in $O(\log n)$ worst-case time per insertion and deletion.

4 A 2-Sided Structure of $O(n)$ Space

In this subsection, we improve the space of our two-sided structure to linear using a two-level sampling idea.

Structure. Let s be an integer between $\log_2 n - 1$ and $\log_2 n + 1$. We divide the domain $\mathbb{R}$ into a set $\mathcal{I}$ of $g = \Theta(n/\log n)$ disjoint intervals $\mathcal{I}_1, ..., \mathcal{I}_g$ such that each $\mathcal{I}_i$ ($1 \leq i \leq g$) covers between $s/2$ and s points of P. Define $\mathcal{C}_i = \mathcal{I}_i \cap P$, and call it a *chunk*. Store the points of each $\mathcal{C}_i$ in an array (i.e., one array per chunk).

We build a structure T of Lemma 3 on $\{\mathcal{I}_1, ..., \mathcal{I}_g\}$. T allows us to sample at the chunk level, when given a query range $q^* = [x^*, y^*]$ aligned with the intervals' endpoints (in other words, q^* equals the union of several consecutive intervals in $\mathcal{I}$). More specifically, given a query with such a range q^* and parameter t, we can use T to obtain a sequence S of t chunk ids, each of which is taken uniformly at random from the ids of the chunks whose intervals are covered by q^*. We slightly augment T such that whenever a chunk id i is returned in S, the chunk size $|\mathcal{C}_i|$ is always returned along with it. The space of T is $O(g \log g) = O(n)$.

We will also need a rank structure on P, which (as explained in Section 1) allows us to obtain t samples from any query range in $O(t \log n)$ time.

Query. We answer a query with parameters $q = [x, y]$ and t as follows. First, in $O(\log n)$ time, we can identify the intervals $\mathcal{I}_i$ and $\mathcal{I}_{i'}$ that contain x and y, respectively. If $i = i'$, we answer the query bruteforce by reading all the $O(\log n)$ points in $\mathcal{C}_i$.

If $i \neq i'$, we break q into three disjoint intervals $q_1 = [x, x^*]$, $q_2 = [x^*, y^*]$, and $q_3 = [y^*, y]$, where x^* (y^*, resp.) is the right (left, resp.) endpoint of $\mathcal{I}_i$ ($\mathcal{I}_{i'}$, resp.). In $O(\log n)$ time (using the rank structure on P), we can obtain the number of data points in the three intervals: $k_1 = |q_1 \cap P|$, $k_2 = |q_2 \cap P|$, and $k_3 = |q_3 \cap P|$. Let $k = k_1 + k_2 + k_3$.

We now determine the numbers t_1, t_2, t_3 of samples to take from q_1, q_2, and q_3, respectively. To do so, generate t random integers in $[1, k]$; t_1 equals how many of those integers fall in $[1, k_1]$, t_2 equals how many in $[k_1 + 1, k_1 + k_2]$, and t_3 how many in $[k_1 + k_2 + 1, k]$. We now proceed to take the desired number of samples from each interval (we will clarify how to do so shortly). Finally, we randomly permute the t samples in $O(t)$ time, and return the resulting permutation.

Sampling t_1 and t_3 elements from q_1 and q_3 respectively can be easily done in $O(\log n)$ time. Next, we concentrate on taking t_2 samples from q_2. If $t_2 \leq 6 \ln 2$, we simply obtain t_2 samples from the rank structure in $O(t_2 \log n) = O(\log n)$ time. For $t_2 > 6 \ln 2$, we first utilize T to obtain a sequence S of $4t_2$ chunk ids for the range $q_2 = [x^*, y^*]$. We then generate a sequence $\mathcal{R}$ of samples as follows. Take the next id j from S. Toss a coin with head probability $|\mathcal{C}_j|/s$.[2] If the coin tails, do nothing; otherwise, append to $\mathcal{R}$ a point selected uniformly at random from $\mathcal{C}_j$. The algorithm finishes as soon as $\mathcal{R}$ has collected t_2 samples. It is possible, however, that the length of $\mathcal{R}$ is still less than t_2 even after having processed all the $4t_2$ ids in S. In this case, we restart the *whole* query algorithm from scratch.

We argue that the expected cost of the algorithm is $O(\log n + t)$. As $|\mathcal{C}_j|/s \geq 1/2$ for any j, the coin we toss in processing S heads at least $4t_2/2 = 2t_2$ times in expectation. A simple application of Chernoff bounds shows that the probability it heads less than t_2 times is at most $1/2$ when $t_2 > 6 \ln 2$. This means that the algorithm terminates with probability at least $1/2$. Each time the algorithm is repeated, its cost is bounded by $O(\log n + t)$ (regardless of whether another round is needed). Therefore, overall, the expected running time is $O(\log n + t)$.

Update. T is updated whenever a chunk (either its interval or the number of points therein) changes. This can be done in $O(\log n)$ time per insertion/deletion of a point in P. A chunk overflow (i.e., size over s) or underflow (below $s/2$) can be treated in $O(s)$ time by a chunk split or merge, respectively. Standard analysis shows that each update bears only $O(1)$ time amortized. Finally, to make sure s is between $\log_2 n - 1$ and $\log_2 n + 1$, we rebuild the whole structure whenever n has doubled or halved, and set $s = \log_2 n$. Overall, the amortized update cost is $O(\log n)$. The amortization can be removed by standard techniques [13]. We have now established:

Theorem 4: For the IRS problem, there is a RAM structure of $O(n)$ space that can answer a two-sided WR query in $O(\log n + t)$ expected time, and can be updated in $O(\log n)$ worst-case time per insertion and deletion.

5 Reduction from WoR to WR

We will need the fact below:

Lemma 5: Let S be a set of k elements. Consider taking $2s$ samples uniformly at random from S with replacement, where $s \leq k/(3e)$. The probability that we get at least s distinct samples is at least $1/2$.

Proof: Denote by R the set of samples we obtain after eliminating duplicates. Consider any $t < s$, and an arbitrary subset S' of S with $|S'| = t$. Thus, $\mathbf{Pr}[R \subseteq S'] = (t/k)^{2s}$. Hence, the probability that $R = S'$ is at

[2] This can be done without division: generate a random integer in $[1, s]$ and check if it is smaller than or equal to $|\mathcal{C}_j|$.

most $(t/k)^{2s}$. Therefore:

$$\begin{aligned}
\mathbf{Pr}[|R| < s] &= \sum_{t=1}^{s-1} \mathbf{Pr}[|R| = t] \\
&\leq \sum_{t=1}^{s-1} \binom{k}{t} (t/k)^{2s} \\
&\leq \sum_{t=1}^{s-1} (ek/t)^t \cdot (t/k)^{2s} \\
(\text{by } e^t < e^s < e^{2s-t}) &< \sum_{t=1}^{s-1} (et/k)^{2s-t} \\
&< \sum_{t=1}^{s-1} (es/k))^{2s-t} \\
&\leq \frac{es/k}{1 - es/k} \leq \frac{1/3}{2/3} = 1/2.
\end{aligned}$$

The lemma thus follows. □

A two-sided WoR query with parameters q, t on dataset P can be answered using a structure of Theorem 4 as follows. First, check whether $t \geq k/(3e)$ where $k = |P(q)|$ can be obtained in $O(\log n)$ time. If so, we run a sampling WoR algorithm (e.g., [15]) to take t samples from $P(q)$ directly, which requires $O(\log n + k) = O(\log n + t)$ time. Otherwise, we run a WR query with parameters $q, 2t$ to obtain a sequence $\mathcal{R}$ of samples in $O(\log n + t)$ expected time. If $\mathcal{R}$ has at least t distinct samples (which can be checked in $O(t)$ expected time using hashing), we collect all these samples into a set S, and sample WoR t elements from S; the total running time in this case is $O(\log n + t)$. On the other hand, if $\mathcal{R}$ has less than t distinct elements, we repeat the above by issuing another WR query with parameters $q, 2t$. By Lemma 5, a repeat is necessary with probability at most $1/2$. Therefore, overall the expected query time remains $O(\log n + t)$.

Similarly, a one-sided WoR query can be answered using a structure of Theorem 1 in $O(\log \log n + t)$ expected time.

6 Concluding Remarks and Future Work

In this paper, we have described an IRS structure that consumes $O(n)$ space, answers a WoR/WR query in $O(\log n + t)$ expected time, and supports an insertion/deletion in $O(\log n)$ time. The query time can be improved to $O(\log \log n + t)$ if the query range is one-sided.

For two-sided queries, we can make the query time $O(\log n + t)$ hold deterministically (currently, it is expected). For that purpose, we need sophisticated big-twiddling structures (specifically, the *fusion tree* [5]) that allow one to beat comparison-based lower bounds on searching an ordered set. We do not elaborate further on this because the resulting structures are perhaps too complex to be interesting in practice. How to make the one-sided query time $O(\log \log n + t)$ hold deterministically is still an open question. IRS has also been studied in external memory (i.e., in the scenario where P does fit in memory). Interested readers may refer to [9] for details.

The concept of *independent query sampling* can be integrated with any reporting queries (e.g., multidimensional range reporting, stabbing queries on intervals, half-plane reporting, etc.), and defines a new variant for every individual problem. All these variants are expected to play increasingly crucial roles in countering the

big-query issue. The techniques developed in this paper pave the foundation for further studies in this line of research.

Acknowledgments

This work was supported in part by projects GRF 4164/12, 4168/13, and 142072/14 from HKRGC.

References

[1] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. Congressional samples for approximate answering of group-by queries. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 487–498, 2000.

[2] Lars Arge and Jeffrey Scott Vitter. Optimal external memory interval management. *SIAM Journal of Computing*, 32(6):1488–1508, 2003.

[3] Vladimir Braverman, Rafail Ostrovsky, and Carlo Zaniolo. Optimal sampling from sliding windows. *Journal of Computer and System Sciences (JCSS)*, 78(1):260–272, 2012.

[4] Pavlos Efraimidis and Paul G. Spirakis. Weighted random sampling with a reservoir. *Information Processing Letters (IPL)*, 97(5):181–185, 2006.

[5] Michael L. Fredman and Dan E. Willard. Blasting through the information theoretic barrier with fusion trees. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 1–7, 1990.

[6] Rainer Gemulla, Wolfgang Lehner, and Peter J. Haas. Maintaining bernoulli samples over evolving multisets. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 93–102, 2007.

[7] Jens Gustedt. Efficient sampling of random permutations. *Journal of Discrete Algorithms*, 6(1):125–139, 2008.

[8] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 171–182, 1997.

[9] Xiaocheng Hu, Miao Qiao, and Yufei Tao. Independent range sampling. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 246–255, 2014.

[10] Chris Jermaine, Subramanian Arumugam, Abhijit Pol, and Alin Dobra. Scalable approximate query processing with the dbo engine. *ACM Transactions on Database Systems (TODS)*, 33(4), 2008.

[11] Suman Nath and Phillip B. Gibbons. Online maintenance of very large random samples on flash storage. *The VLDB Journal*, 19(1):67–90, 2010.

[12] Frank Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[13] Mark H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, 1983.

[14] Abhijit Pol, Christopher M. Jermaine, and Subramanian Arumugam. Maintaining very large random samples using the geometric file. *The VLDB Journal*, 17(5):997–1018, 2008.

[15] Jeffrey Scott Vitter. Random sampling with a reservoir. *ACM Trans. Math. Softw.*, 11(1):37–57, 1985.

[16] Zhewei Wei and Ke Yi. Beyond simple aggregates: indexing for summary queries. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 117–128, 2011.

Hidden Database Research and Analytics (HYDRA) System

Yachao Lu[†], Saravanan Thirumuruganathan[‡], Nan Zhang[†], Gautam Das[‡]
The George Washington University[†], University of Texas at Arlington[‡]
{yachao, nzhang10}@gwu.edu[†], {saravanan.thirumuruganathan@mavs, gdas@cse}.uta.edu[‡]

Abstract

A significant portion of data on the web is available on private or hidden databases that lie behind form-like query interfaces that allow users to browse these databases in a controlled manner. In this paper, we describe System HYDRA that enables fast sampling and data analytics over a hidden web database with a form-like web search interface. Broadly, it consists of three major components: (1) SAMPLE-GEN which produces samples according to a given sampling distribution (2) SAMPLE-EVAL that evaluates samples produced by SAMPLE-GEN and also generates estimations for a given aggregate query and (3) TIMBR that enables fast and easy construction of a wrapper that models both input and output interface of the web database thereby translating supported search queries to HTTP requests and retrieving top-k query answers from HTTP responses.

1 Introduction

1.1 Motivation

A large portion of data available on the web is present in the so called "Deep Web". The deep web consists of private or hidden databases that lie behind form-like query interfaces that allow users to browse these databases in a controlled manner. This is typically done by search queries that specify desired (ranges of) attribute values of the sought-after tuple(s), and the system responds by returning a few (e.g., top-k) tuples that satisfy the selection conditions, sorted by a suitable ranking function. There are numerous examples of such databases, ranging from databases of government agencies (such as data.gov, cdc.gov), databases that arise in scientific and health domains (such as Pubmed, RxList), to databases that occur in the commercial world (such as Amazon, EBay).

While hidden database interfaces are normally designed to allow users to execute search queries, for certain applications it is also useful to perform *data analytics* over such databases. The goal of data analytics is to reveal insights and "big picture" information about the underlying data. In particular, we are interested in data analytics techniques that can be performed only using the public interfaces of the databases while respecting the data access limitations (e.g., query rate limits) imposed by the data owners. Such techniques can be useful in powering a multitude of third-party applications that can effectively function anonymously without having to enter into complex (and costly) data sharing agreements with the data providers. For example, an economist would be interested in tracking economically important aggregates such as the count of employment postings in various categories every month in job search websites.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

In the literature, the three common data analytics tasks have been *crawling*, *sampling*, and *aggregates estimation*. The objective of crawling is to retrieve all tuples from the database. The tuples thus collected can serve as a local repository over which any analytics operation can be performed. Unlike crawling, sampling aims to collect only a small subset of tuples of the database drawn randomly according to a pre-determined sampling distribution. This distribution can be uniform (leading to a simple random sample) or can be weighted. If collected appropriately, the sample is representative of the database and can therefore be used for a wide variety of analytics purposes. Aggregate estimation is the final task which aims to answer aggregate queries (AVG, SUM, COUNT, etc.) over the database, so as to enable data analytics and mining applications that can be built upon the answered aggregates. Aggregate estimation is better suited when we already know which aggregate needs to be estimated (thereby allowing us to tailor the analytics process for the specific aggregate in question), while sampling is more flexible but may not be as efficient for estimating a given aggregate. All of these tasks may be carried out over a single snapshot of a hidden database or continuously run when the database changes over time.

Today, data analytics over hidden databases face the following significant challenges:

- **No direct way of executing data analytics queries:** The input search interface of a hidden database is usually a web form, allowing a user to only formulate simple queries, such as conjunctive queries or kNN nearest neighbor queries. There is no direct way of specifying aggregate or data analytics queries. Likewise, the returned results are revealed via an output interface that limits the number of tuples returned (the *top-k constraint*) ordered by an often-proprietary ranking function. This makes data analytics problems such as sampling difficult, as the returned tuples for a query are not necessarily representative samples of the underlying database, e.g., there may be a bias towards tuples "favored" by the ranking function used by the website - which for a e-commerce website could mean products with lower prices, higher customer ratings, etc.

- **Query rate Limitations:** Most hidden databases limit how many search queries (essentially HTTP requests) a user (or an IP address, an API account, etc.) can issue over a given time period. Search APIs are limited to 5000 queries per day in EBay or 180 queries every 15 minutes in Twitter. Since search queries form the only access channel we have over a hidden database, this limit requires us to somehow translate any data analytics task into a small number of search queries, feasible to issue within a short time period even under the query rate limitation. The bad news is that the query rate requirement for the data analytics task of *crawling* is too large to be practical under the query rate limitation enforced by real-world hidden databases.Therefore, crawling is not a focus of this paper, and we henceforth only consider sampling and aggregate estimation.

- **Understanding the query interface:** Thus far in our discussions, there is an implicit assumption that the query interface of a hidden database has already been "understood", i.e., a site-specific wrapper has already been designed to enable mapping of user specified queries into the web form and to retrieve tuples from the returned result page. In reality, this is an extremely difficult task to automate for arbitrary websites as well to maintain the wrappers when the interface designs are updated by the websites. Understanding query interfaces and wrapper generation are well-recognized research problems in the area of information extraction.

1.2 System Hydra

We develop System Hydra (Hidden Database Research and Analytics) which addresses the above challenges of data analytics over hidden databases. It consists of three components: two main components SAMPLE-GEN and SAMPLE-EVAL and one auxiliary component TIMBR. SAMPLE-GEN enables the user to specify a sampling distribution and then obtain samples according to the specified distribution. SAMPLE-EVAL, on the

other hand enables the estimation of a user-specified aggregate query. If the hidden database of concern provides a standardized search query interface, then the TIMBR component is not really needed. Otherwise, if only a form-like web interface is provided by the hidden database, the user can use the TIMBR component to easily build a wrapper that translates the search queries required by the two main components to HTTP requests and responses to and from the hidden database server.

The main technical contributions of the system can be summarized as follows:

- For the generation of samples in SAMPLE-GEN, our focus is to enable the generation of samples with statistical guarantees while minimizing the number of HTTP requests made to the target hidden database. The motivation here is to avoid overloading the target server, and also because many real-world hidden databases limit the number of requests that can be made from an IP address or user/API account for a given time period. The main techniques in SAMPLE-GEN include two parts: One handles traditional SQL-like query semantics - i.e., the tuples returned must match the selection conditions specified in the input query. The other handles kNN query semantics - i.e., while the returned tuples may not match all selection conditions, they are ordered according to a pre-defined, often proprietary, distance function with the specified conditions. We found that not only these two query-return semantics call for the design of different sampling techniques, there are many other subtler interface issues that affect sampling design - e.g., when the returned results are truncated to the top-k tuples, whether the real COUNT of matching tuples (for the first SQL-like semantics) is also returned. We shall discuss these subtle issues and implications in detail in the paper.

- For the estimation of aggregates in SAMPLE-EVAL, the main focus is on understanding how the generation of (certain) samples affect the end goal, which is often to estimate one or more aggregate queries. The design of SAMPLE-EVAL considers three key factors associated with the usage of a sample tuple for the purpose of aggregate estimations: (1) bias, i.e., whether the expected value of an aggregate estimation (from samples) agrees with its real value; (2) variance, i.e., how adjusting the sampling distribution affects the variance of estimated aggregates, which is as important as bias given that the mean square error of an estimation is the SUM of bias^2 and variance; and (3) longevity, i.e., when the underlying database changes, whether a sample tuple needs to be frequently updated, or remains useful for a long time. We shall discuss in the paper how SAMPLE-EVAL evaluates the three facets and determines how to adjust the sampling process based on the intended application (e.g., aggregates to be estimated) and the information learned so far from the hidden database.

- For understanding the input interface, TIMBR provides two methods: (1) a user can click on any input control (e.g., text-box, dropdown menu) and map it to an attribute of interest, and (2) for websites with more complex designs (e.g., calendar inputs which are not standard HTML components but manually implemented using HTML, CSS and JavaScript), TIMBR allows a user to train the system by recording a sequence of interactions with the web page and mapping the sequence to an input value. For understanding the output interface, TIMBR once again provides two methods to accommodate the different design complexities of webpages for real-world hidden databases. For most websites, the human interaction is as simple as clicking on a tuple of interest and then click on the attributes of interest within the tuple. The TIMBR component can then automatically find other tuples and their corresponding values on the attributes of interest. There are a small number of websites, however, that call for a more subtle design. For example, some websites mix query answers with promotional results (e.g., tuples that do not match the input query). To properly define the results of interest for these websites, TIMBR also provides a method for a human user to specify not one, but two tuples of interest. Our system then automatically learn the differentiator between the tuples of interest and (potentially) other tuples displayed on the web page, enabling the precise selection of returned tuples.

It is important to understand that our goal here is not to develop new research results for the vast field of information extraction, but to devise a simple solution that fits our goal of the HYDRA system - i.e., a tool that a data analyst with substantial knowledge of the underlying data (as otherwise he/she would not be able to specify the aggregate queries anyway) can quickly deploy over a form-like hidden database.

The rest of this paper is organized as follows. Section 2 introduces the data model and a taxonomy of data analytics over hidden databases. Section 3 provides an overview of the system HYDRA and its various components that are described in following sections. Section 4 describes key techniques used by the first component - SAMPLE-GEN - to retrieve samples from underlying hidden web databases. Section 5 focusses on component SAMPLE-EVAL that evaluates the samples collected by SAMPLE-GEN. Section 6 describes the TIMBR component for modeling the input and output interface of a hidden web databases. We describe the related work in Section 7 followed by final remarks and future work in Section 8.

2 Preliminaries

2.1 Model of Hidden Databases

Consider a hidden web database D with n tuples and m attributes $A_1, \ldots, A_m$, with the domain of attribute A_i being U_i, and the value of A_i for tuple t being $t[A_i]$. Reflecting the setup of most real-world hidden databases, we restrict our attention to categorical (or ordinal) attributes and assume appropriate discretization of numeric attributes (e.g., price into ranges such as (0, 50], (50, 100], etc., as in hidden databases such as amazon.com).

Input Search Interface: The search interface of a hidden database is usually formulated like a web form (see Figure 1 for an example), allowing a user to specify the desired values on a subset of attributes, i.e., $A_{i_1} = v_{i_1}$ & $\ldots$ & $A_{i_s} = v_{i_s}$, where $i_1, \ldots, i_s \in [1, m]$ and $v_{i_j} \in U_{i_j}$. In Figure 1, the user specifies a query to return cars with predicates `Make=Ford & Model=F150 & ... & ForSaleBy=All Sellers`. While some hidden databases allow any arbitrary subset of attributes to be specified, many others place additional constraints on which subsets can or cannot be specified. For example, most hidden databases require at least one attribute value to be specified, essentially barring queries like `SELECT * FROM` D. Some even designate a subset of attributes to be "required fields" in search (e.g., departure city, arrival city and departure date in a flight search database).

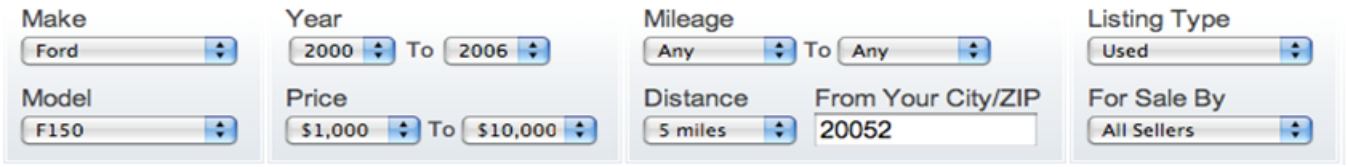


Figure 1: A form-like input interface for a Hidden Database

Another, somewhat subtler, constraint is what we refer to as the "auto-filling searches" - i.e., the hidden database might use information specified in a user's profile to "auto-fill" the desired value for an attribute instead of including the attribute in the web form (or when the user leaves the attribute as blank in search). This constraints most often present in hidden databases with social features - dating websites such as match.com and online social networks such LinkedIn fall into this category. With these hidden databases, when a user searches for another one to connect to, the database often use the profile information of the searching user, sometimes without explicitly stating so in the search interface.

Output Interface: A common property shared by the output interfaces of almost almost all hidden databases is a limit on the number of tuples returned (unlike traditional databases where the query returns all matching

tuples by default). By the very nature of HTTP-based transmission, the number of tuples loaded at once has to be limited to ensure a prompt response. Of course, a hidden database may support "page down" or "load more" operations that retrieve additional batches of tuples - but the number of such operations allowed for a query is often limited as well (e.g., amazon.com limits the number of page downs to 100). As such, we say that a hidden database enforces a *top-k constraint* when it restricts each query answer to at most k tuples, which can be loaded at once or in multiple batches.

In terms of which (up-to-) k tuples to return, note from the above discussions that we refer to the user-specified conditions as "desired values" rather than "equality predicates". The reason for doing so is because many hidden databases do not limit the returned results to tuples that exactly match all user-specified desired values. Indeed, we can broadly classify hidden database output semantics into two categories, *exact match* and kNN, based on how a user query is answered.

In the exact match scenario, for a given user query q, the hidden database first finds all tuples matching q, denoted as $Sel(q) \subseteq D$. Then, if $|Sel(q)| \leq k$, it simply returns $Sel(q)$ as the query answer. In this case, we say that the query answer is *valid* if $0 < |Sel(q)| \leq k$, or it *underflows* if $Sel(q) = \Omega$. When $|Sel(q)| > k$, however, we say that an *overflow* occurs, and the database has to rely on a proprietary *ranking function* to select k tuples from $Sel(q)$ and return them as the query answer, possibly accompanied by an *overflow flag* - e.g., a warning message such as "your query matched more than k tuples, or an informative message such as "your query matched $|Sel(q)|$ tuples, but only k of them are displayed").

The ranking function used in this exact match scenario may be static or query dependent. Specifically, note that a ranking function can be represented as a scoring function $f(t, q)$ - i.e., for a given query q, the k tuples t with the highest $f(t, q)$ are returned. Here a ranking function is *static* if for a given tuple t, $f(t, q)$ is constant for all queries. An example here is any ranking by a certain attribute - e.g., by price in real-world e-commerce websites. On the other hand, a ranking function is query-dependent if $f(t, q)$ varies for different q - e.g., in flight search, rank by duration if q contains CLASS = BUSINESS, and rank by price if CLASS = ECONOMY.

In the kNN scenario, the hidden database can be considered as simply ranking *all* tuples by a query-dependent ranking function $f(t, q)$ - which essentially (inversely) measures the distance between t and q - and returns the k tuples with the highest $f(t, q)$ (i.e., shortest distances). One can see that, clearly, the returned tuples might not exactly match q on all attributes specified.

Query Rate Limitations: The most critical constraint enforced by hidden databases, impeding our goal of enabling data analytics, is the *query rate* limitation - i.e., a limit on how many search queries a user (or an IP address, an API account, etc.) can issue over a given time period. For example, ebay.com limits the number of queries per IP address per hour to 5000. Since search queries form the only access channel we have over a hidden database, this limit requires us to somehow translate any analytics operations we would like to enable into a small number of search queries, feasible to issue within a short time period even under the query rate limitation.

2.2 Data Analytics over Hidden Databases

In the literature, three common operations over hidden databases have been studied: crawling, sampling and data analytics. The objective of *crawling* is to retrieve all tuples from D. The tuples thus collected can serve as a local repository over which any analytics operation can be performed. The good news about crawling is that its performance boundaries are well understood - [1] reports matching lower and upper bounds (upper to a constant difference) on the number of queries required for crawling a hidden database. The bad news, however, is even the lower bound is too large to be practical under the query rate limitation enforced by real-world hidden databases. As such, crawling is not a focus of the HYDRA system.

Unlike crawling, *sampling* aims to collect only a small subset of tuples $S \subset D$ with $|S| \ll |D|$ - the challenge here is that the subset must be drawn randomly according to a pre-determined sampling distribution. This distribution can be uniform (leading to a simple random sample) or can be weighted based on certain

attribute values of interest (leading to a weighted sample). If collected appropriately, the sample is representative of D and can therefore be used for a wide variety of analytics purposes.

Data analytics is the final task which aims to answer aggregate queries (AVG, SUM, COUNT, etc.) over D, so as to enable data mining applications that can be built upon the answered aggregates. As discussed in the introduction, data analytics is better suited when we already know the exact aggregates to be estimated, while sampling is more flexible but may not be as efficient for estimating a given aggregate.

All of these tasks may be carried out over a single snapshot of a hidden database or continuously run when the database changes over time. In the continuous case, the sampling task would need to maintain an up-to-date sample of the current database, while the data analytics task would need to continuously monitor and update the aggregate query answers. In addition, the data analytics task may also need to consider aggregates *across* multiple versions of the hidden database - e.g., the average price cut that happened during the Thanksgiving holiday for all products at a e-commerce website.

Performance Measures: Efficiency-wise, the bottleneck for all the three tasks centers on their *query cost*, i.e., the number of search queries one needs to issue to accomplish a task, because of the aforementioned query rate limitation.

Besides the efficiency measure, sampling and data analytics tasks should also be measured according to the *accuracy* of their outputs. For sampling, the accuracy measure is *sample bias* - i.e. the "distance" between the pre-determined (target) sampling distribution and the actual distribution according to which sample tuples are drawn.

For data analytics, since aggregate query answers, say θ are often estimated by randomized estimators $\tilde{\theta}$, the accuracy is often measured as the expected distance between the two, e.g., the mean squared error (MSE) $MSE(\tilde{\theta}) = E[(\tilde{\theta} - \theta)^2]$, where $E[\cdot]$ denotes the expected value taken over the randomness of $\tilde{\theta}$. Here the MSE actually consists of two components, *bias* and *variance*. Specifically, note that

$$MSE(\tilde{\theta}) = E[(\tilde{\theta} - \theta)^2] = E[(\tilde{(\theta)} - E(\theta))^2] + (E[\tilde{\theta}] - \theta)^2 = Var(\tilde{\theta}) + Bias^2(\tilde{\theta}). \quad (13)$$

An estimator is said to be biased if $E[\tilde{\theta}] \neq \theta$. In contrast, the mean estimate of an unbiased estimator is equal to the true value. The variance of the estimator determines the spread of the estimation (or how the estimate varies from sample to sample). Of course, the goal here is to have unbiased estimators with small variance.

3 Overview of Hydra

Figure 1 depicts the architecture of System HYDRA. It consists of three components: two main components SAMPLE-GEN and SAMPLE-EVAL and one auxiliary component TIMBR. From the perspective of a HYDRA user, the three components work as follows. SAMPLE-GEN enables the user to specify a sampling distribution and then obtain samples according to the specified distribution. SAMPLE-EVAL, on the other hand enables the estimation of a user-specified aggregate query. If the hidden database of concern provides a standardized search query interface, e.g., search APIs such as those provided by Amazon or EBay, then the TIMBR component is not really needed. Otherwise, if only a form-like web interface is provided by the hidden database, the user can use the TIMBR component to easily build a wrapper that translates the search queries required by the two main components to HTTP requests and responses to and from the hidden database server.

Internally, these three components interact with each other (and the remote hidden database) in the following manner. SAMPLE-GEN is responsible for most of the heavy-lifting - i.e., taking a sampling distribution as input and then issuing a small number of search queries to the hidden database in order to produce sample tuples as output. What SAMPLE-EVAL does is to take an aggregate query as input and then select an appropriate sampling distribution based on both the aggregate query and the historic samples it received from SAMPLE-GEN. Once SAMPLE-EVAL receives the sample tuples returned by SAMPLE-GEN, it generates an estimation for the aggregate query answer and then return it to the user. As mentioned above, TIMBR is an optional component

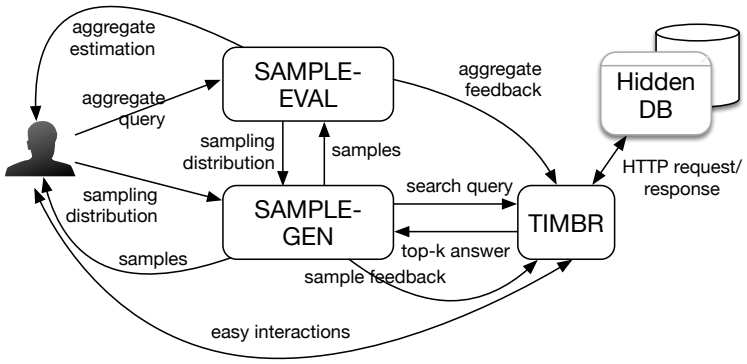


Figure 2: Architecture of HYDRA System

that bridges SAMPLE-GEN and the hidden database server by translating a search query to one or more HTTP requests, and HTTP responses back to the top-k query answers. To do so in an efficient manner, TIMBR solicits easy-to-specify, mouse-click-based, inputs from the user, as well as sample and aggregate feedback from SAMPLE-GEN and SAMPLE-EVAL, respectively. For example, to properly understand the domains of various attributes, it may extract domain values by drawing sample feedback from SAMPLE-GEN, and estimate the popularity of a domain value by drawing aggregate feedback from SAMPLE-EVAL.

4 Component SAMPLE-GEN

In this section, we briefly summarize the key techniques used by SAMPLE-GEN to retrieve samples from underlying hidden web databases. As described in Section 2, we highlight the two most common return semantics used in hidden databases - exact match and kNN. These two models require substantially different techniques. Nevertheless, the estimators used are unbiased, has reasonable query cost and often also has low variance.

4.1 Sample Retrieval for Matching Tuples

We start by describing various techniques to obtain unbiased samples over a hidden databases with a conjunctive query interface and an exact match return semantics. For ease of exposition, we consider a static hidden database with all Boolean attributes and that our objective is to obtain samples with uniform distribution. Please refer to [2, 3] for further generalization and additional optimizations. All the algorithm utilize an abstraction called query tree.

Query Tree: Consider a tree constructed from an arbitrary order of the input attributes $A_1, \ldots, A_m$. By convention, we denote the root as Level 1 and it represents the query `SELECT * FROM D`. Each internal (non-leaf) node at level i are labelled with attribute A_i and has exactly 2 outgoing edges (in general U_i edges) one labelled 0 and another labelled 1. Note that each path from the root to a leaf represents a particular assignment of values to attributes. The attribute value assignments are obtained from the edges and each leaf representing potential tuples. In other words, only a subset of the leaf nodes correspond to valid tuples. Figure 3 displays a query tree.

Brute-Force-Sampler: We start by describing the simplest possible sampling algorithm. This algorithm constructs a fully specified query q (i.e. a query containing predicates for all attributes) as follows: for each attribute A_i, it chooses the value v_i to be 0 with probability 0.5 and 1 with probability 0.5. The constructed query is then issued with only two possible outcomes - valid or underflow (since all tuples are distinct). If a tuple is returned, it is picked as a sample. This process is repeated till the requisite number of samples are

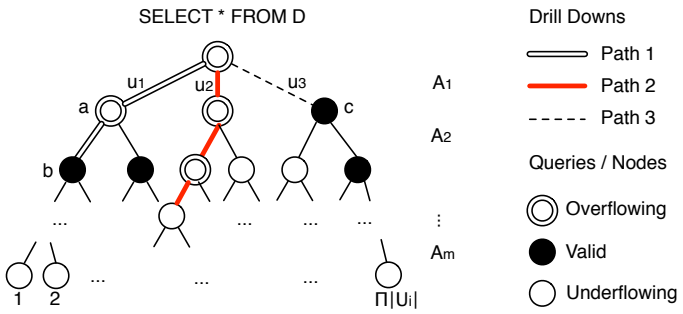


Figure 3: Query Tree

obtained. Notice that this process is akin to picking a leaf node from the query tree uniformly at random. We can see that this process generates unbiased samples. However, this approach is prohibitively expensive as the number of tuples in the database is much smaller than the cartesian product of the attribute values - in other words $n \ll 2^m$. Hence most of the queries will result in underflow necessitating large number of queries to obtain sufficient samples.

Hidden-DB-Sampler: Hidden-DB-Sampler, proposed in [4], takes a slightly more efficient approach. Instead of directly issuing fully specified queries, this sampler performs a fundamental operation known as *random drilldown*. It starts by issuing the query corresponding to the root. At level i, it randomly picks one of the values $v_i \in U_i$ corresponding to attribute A_i and issues the query corresponding to that internal node. Notice that if the query underflows, then the drilldown can be immediately terminated (as further queries will also return empty). If the query is valid, then one of the returned tuples is randomly picked as a sample. If the query overflows, the process is continue till a valid internal or leaf node is reached. We can see that this sampler requires less query cost than Brute-Force-Sampler. However, the efficiency comes at the cost of skew where the tuples no longer are reached with same probability. Specifically, note that "short" valid queries are more likely to be reached than longer queries. A number of techniques such as acceptance/rejection sampling could be used to fix this issue. The key idea is to accept a sample tuple with probability proportional to $1/2^l$ where l is the length/number of predicates in the query issued. However, it is possible that a number of tuples could be rejected before a sample is accepted.

Samplers that Leverage Count Information: As pointed out in Section 2, most real-world hidden web databases provide some feedback about whether a query overflows by either alerting the user or providing the number of matching tuples for each query. It is possible to leverage the count information to design more efficient samplers.

Let us first consider the scenario where the count information is available. Count-Decision-Tree-Sampler, proposed in [5], uses two simple ideas to improve the efficiency of Hidden-DB-Sampler. First, it leverages *query history* - the log of all queries issued, their results and the counts. This information is then used in preparing which query to execute next. Specifically, given a set of candidate queries to choose from, the sampler prefers the query that already appears in the history which results in substantial query savings. Another idea is to generalize the notion of attribute ordering used in query tree. While the previous approaches used an arbitrary but fixed ordering, additional efficiency can be obtained by incrementally building the tree based on a saving function that helps us choose queries that have a higher chance of reaching a random tuple soon. The algorithm for retrieving unbiased samples where only an alert is available is bit trickier. Alert-Hybrid-Sampler operates in two stages. First, it obtains a set of *pilot* samples that is used to estimate the count information for queries. In the second stage, we reuse the algorithm that leverages count information. While this results in an inherent approximation and bias, it is often substantially more efficient in terms of number of queries.

HD-Unbiased-Sampler: This sampler, proposed in [2], is one of the state of the art samplers that uses

a number of novel optimizations for better efficiency. We discuss the fundamental idea here and the various variance reduction techniques in Section 5. In contrast to other techniques that obtain uniform random samples with unknown bias, this sampler intentionally seeks biased samples - albeit those for which the bias is precisely known. This additional knowledge allows us to "correct" the bias and obtain unbiased samples.

The random drilldown process is very similar to the previous samplers. We start with the root node and choose an edge uniformly at random. If the query overflows, the process is continued. If it is valid, we can terminate the process by choosing a tuple uniformly at random from the results. When the query underflows, instead of restarting the drill down, this sampler instead backtracks. Specifically, it backtracks to a sibling node (i.e. a node with which it shares the parent node) and continue the drill down process. We can note that the drill down always terminates with a node whose parent is overflowing. By using few additional queries, [2] proposed an efficient estimator that can precisely compute the probability that an overflowing node is visited. If the ultimate purpose is to estimate aggregates, one can use Horvitz-Thompson estimator to generate estimates based on the known bias of each tuple.

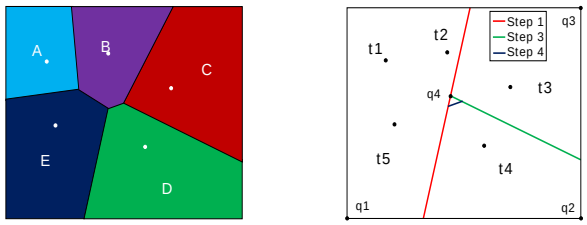
4.2 Sample Retrieval for kNN Model

In this subsection, we examine the kNN return semantics where we are given an arbitrary query q and the system returns k tuples that are nearest to q based on a scoring (or distance) function. The objective is to design efficient techniques for obtaining uniform random samples given a kNN query interface. We focus our attention to a class of hidden databases (spatial databases) and describe recent attempts to retrieve uniform samples. Specifically, we focus on *location based services* (LBS) that have become popular in the last few years. Websites such as Google Maps and other social networks such as WeChat extensively use this return semantics. Given a query point (location) q, an LBS returns k points from D that are closest to q. Note that the hidden database consists of 2-dimension points represented as latitude/longitude which can correspond to points of interest (such as in Google Maps) or user locations (such as in WeChat, Sina Weibo etc). For this article, we assume that the distance function used in Euclidean distance and that the objective is to obtain uniform samples.

Taxonomy of LBS: Based on the amount of information provided for each query, each LBS can be categorized into two classes. A Location-Returned-LBS (LR-LBS) returns the precise location for each of the top-k returned tuples[6]. A number of examples such as Google Maps, Bing Maps, Yelp follow this model as they display the location information of the POIs such as restaurants. A Location-Not-Returned-LBS (LNR-LBS), on the other hand, does not return tuple locations[6]. This is quite common among social networks with some location based functionality such as Weibo due to privacy concerns. Both these categories require substantially different approaches for generating samples with divergent efficiencies. As we shall see later, the availability of location dramatically simplifies the sampling design.

Voronoi Cells: Assume that all points in the LBS are contained in a bounding box B. For example, all POIs in USA can be considered as located in a bounding box encompassing USA. The Voronoi cell of a tuple t, denoted by $V(t)$ is the set of points on the B-bounded plane that are closer to t than any other tuple in D[7]. In other words, given a Voronoi cell of a tuple t, all points within it return t as the nearest neighbor. We can notice that the Voronoi cells are mutually exclusive and also convex[7]. From the definition, we can notice that the ratio of area of the Voronoi cell to the area of the bounding box B precisely provides the probability that a given tuple is the top ranked result. Figure 4a displays the Voronoi diagram for a small database.

Overview of Approach: At a high level, generating samples for LR-LBS and LNR-LBS follow a similar procedure. In both cases, we generate random point queries, issue them over the LBS and retrieve the top ranked tuple. Note that based on the distribution of underlying tuples, some tuples may be retrieved at a higher rate. For example, a POI in a rural area might be returned as a top result for a larger number of query points than a POI in a densely populated area. Hence, the process of randomly generating points and issuing them results in a biased distribution for the top ranked tuples. However, using the idea from HD-Unbiased-Sampler, we can correct the bias if we know the probability that a particular POI will be returned as the top result. For this purpose, we



(a) (b)

Figure 4: Voronoi Diagram and Illustration of LR-LBS-AGG

use the concept of Voronoi cells[7] that provides a way to compute the bias. [6] proposes methods for precise computation of Voronoi cell of a tuple for LR-LBS. For LNR-LBS, [6] describes techniques to compute Voronoi cell of a tuple to arbitrary precision.

Algorithms LR-LBS-AGG and LNR-LBS-AGG: This algorithm generates samples and uses them for aggregate estimation over LR-LBS such as Google Maps. Note that in this case, the location of the tuple is also returned. The key idea behind computation of the Voronoi cell of a tuple is as follows: if we somehow collect all the tuples with Voronoi cell adjacent to t, then we can precisely compute Voronoi cell of t[7]. So the challenge now reduces to designing a mechanism to retrieve all such tuples. A simple algorithm (see [6] for further optimizations) is as follows. We start with a singleton set D' containing t with its potential Voronoi cell containing the whole bounding box B. We issue queries corresponding to the boundaries of B. If any query returns an unseen tuple, we append it to D', recompute the Voronoi cell and repeat the process. If all the boundary points return a tuple that we have already seen, then we can terminate the process. This simple procedure provides an efficient method to compute the Voronoi cell and thereby the bias of the corresponding tuple. We again use Horvitz-Thompson estimator for estimating the aggregate by assigning unequal weights to tuples based on their area. The key challenge for applying this approach to LNR-LBS is that the location of a tuple is not returned. However, [6] proposes a "binary search" primitive for computing the Voronoi cell of a tuple t. When invoked, this primitive identifies one edge of the Voronoi cell of t to arbitrary precision as required.

Figure 4b provides a simple run-through of the algorithm for a dataset with 5 tuples $\{t_1, \ldots, t_5\}$. Suppose we wish to compute $V(t_4)$. Initially, we set $D' = \{t_4\}$ and $V(t_4) = V_0$, the entire region. We issue query q_1 that returns tuple t_5 and hence $D' = \{t_4, t_5\}$. We now obtain a new Voronoi edge that is the perpendicular bisector between t_4 and t_5. The Voronoi edge after step 1 is highlighted in red. In step 2, we issue query q_2 that returns t_4 resulting in no update. In step 3, we issue query q_3 that returns t_3. $D' = \{t_3, t_4, t_5\}$ and we obtain a new Voronoi edge as the perpendicular bisector between t_3 and t_4 depicted in green. In step 4, we issue query q_4 that returns t_2 resulting in the final Voronoi edge depicted in blue. Further queries over the vertices for $V(t_4)$ does not result in new tuples concluding the invocation of the algorithm.

5 Component SAMPLE-EVAL

In this section, we describe the techniques used in the SAMPLE-EVAL component that is used to evaluate the quality of samples collected by SAMPLE-GEN component. In this article, we focus our evaluation of samples based on the end-goal of aggregate estimation. We consider three major factors: bias, variance and longevity. We start by describing bias and variance in in Section 5.1. Since there exist unbiased estimators for common aggregates, we focus on known techniques to reduce the variance of an estimator. In Section 5.2, we describe a principled method for updating sample when the underlying database changes.

5.1 Static Database: Bias and Variance

Recall from Section 2 that the *bias* of an estimator is the difference between the expected value and the true value of the aggregate. In other words, it provides information about whether the expected value of an aggregate (estimated from the samples) agrees with the actual value. It is preferable for an estimator to be unbiased - i.e. have a bias of 0. The *variance* of an estimator measures how adjusting the sampling distribution affects the variance of estimated aggregates - in other words, how aggregate estimated varies from sample to sample.

If an estimator has no bias, then the long term average of the estimates converges to the true value. If the estimator has a small but known bias, then the estimates converges around the biased value. If the estimator has a low variance, most estimates generated from samples cluster around the expected value of the estimator. Finally, if the estimator has a high variance, the estimates generated can vary widely around the expected value. It is possible to have an estimator with large bias and low variance and vice versa.

Recall from Section 2 that bias and variance are tightly interconnected through the MSE of an estimator. If all things being equal, an unbiased estimator is preferable to a biased estimator. If the estimator is unbiased, then it is preferable to have a low variance. Finally, it might not always be possible to design an unbiased estimator - there are a number of practical estimators that are biased but with a small bias. The relevant trade-off between bias and variance is often specific to the end goal.

There exist a number of estimators that provide unbiased estimation of aggregates over hidden databases [2, 3, 4]. Hence in this section, we focus on some principled mechanisms to reduce the variance. For additional details, please refer to [8].

Acceptance/Rejection Sampling: Recall from Section 4 that Hidden-DB-Sampler sought to improve Brute-Force-Sampler by issuing shorter (in terms of number of predicates) and broader (as measured by the subset of database matched) queries. Basically, the sampler performs a random drill down starting from the root. When it reaches an underflowing node, it restarts the drill down. For overflowing nodes, it continues the drill down process while for a valid node, it takes one of the returned tuples as a sample. We can see that this process introduces a bias towards tuples that are favored by the ranking function (and hence returned by shorter queries). A common technique used to correct this is called acceptance rejection sampling[9]. Instead of unconditionally accepting a tuple from a valid query, we discard the tuples that were obtained from a short random drill down with a higher probability. If done correctly, the rejection results in a set of accepted samples that approximate the uniform distribution.

Weighted Sampling: While acceptance/rejection sampling provides a uniform random samples, they often reject a disproportionate number of samples. A major improvement can be achieved by *weighted sampling*[9] that *accepts* all the tuples from the drill down and instead associates with each of them a weight that is proportional to its selection probability $p(q)$ (probability that the tuple is selected). Now, we can generate a more robust estimate by adjusting each estimate by its selection probability. This is often achieved through the unequal probability sampling estimators such as Horvitz-Thompson[9]. In addition to resulting in substantial savings of query cost (as no sample is rejected), the estimator remains unbiased. Of course, it is now necessary to design additional mechanisms to estimate the selection probability as accurate as possible as it has a disproportionate impact over the estimator accuracy. This approach often works well if the accuracy of answering an aggregate query depends on whether the distribution of selection probability of tuples is aligned with the aggregate to be estimated. Please refer to [10] for additional discussions.

Weight Adjustment: This a popular variance reduction technique that has broad applicability in both traditional and hidden database sampling. The key idea here is to avoid the potential misalignment between aggregate and selection probability distribution by proactively seeking to "align" these factors. Specifically, this is achieved by adjusting the probability for following each branch in a query tree. Such a change affects the selection probability of nodes and potentially results in a better alignment. Achieving perfect alignment is often hard and would make the estimator too specific to an aggregate. Instead, it is possible to design an effective heuristics that has generality and often approximates the perfect alignment with reasonable fidelity. The typical

strategy to perform weight alignment is to obtain a set of "pilot" samples and use them to estimate the measure attribute distribution. From the pilot samples, we can estimate for each branch the selection probability proportional to the size of the database subset rooted under this branch. While knowledge of the exact sizes results in 0 variance, the approximation via pilot samples often reduces the variance in practice. Please refer to [2] for additional discussions.

Crawling of Low-Probability Zones: There exists a number of well known samplers [5] that could leverage overflow information to improve the efficiency of sampling. This is achieved by having a constant cutoff C that trades-off the balance between sampling efficiency and variance. These samplers assume that all queries at Level-C results in underflow or is valid and choose branch with different probabilities. Such an approach has the unfortunate side-effect of rarely choosing tuples that are only returned by leaf level query nodes. This skew between tuples that are reachable at low and high levels can be mitigated by combining sampling with crawling of low probability zones. Specifically, the sampling of tuples below Level-C starts when the sampler reaches an overflow query Q at level C but could not select a sample from it (possibly due to rejection sampling)[10]. Instead of continuing the drill down below level-C, we can crawl the entire sub-tree rooted at Q and use the results to estimate the size of tuples in the sub-tree that match Q. A tuple randomly chosen from the set of tuples that match Q is returned as sample. We can see that this modification reduces the skew of the selection probability of tuples below Level-C.

5.2 Dynamic Databases: Longevity

In this subsection, we briefly discuss the third factor in reducing estimation variance by considering "longevity". The approaches discussed in previous section are applicable only for static databases. However, most real world databases often undergoes updates (inserts, modifications and deletions). Longevity is a key factor in the determining how a sample tuple retrieved using one of the prior rounds must be handled.

A straightforward, but naive, approach for handling dynamic databases is "repeated execution" (RESTART-ESTIMATOR) where samplers for static databases are invoked after each round. However, such an approach has number of issues. First, the query budget for each round is spent wholly on retrieving new samples for that round. This causes the variance of the aggregate estimation in each round to remain more or less constant (as it in turn inversely related to the number of samples). We can readily see that it must be possible to have a better process that carefully apportions the query budget between updating samples from prior rounds and retrieving new samples.

A straightforward improvement is REISSUE-ESTIMATOR[3] that outperforms the naive algorithm by leveraging historic query answers. Specifically, any random drill down conducted is uniquely represented by a "signature". After the first round, the signature and results of the drill down are stored. In the subsequent rounds, we reuse the same set of signatures (as against the naive algorithm that generates new drill downs). Recall that all the samplers for static databases terminated the drill down when they reached a valid node. Hence, in each subsequent round, we can start our process from the signature node. If the node overflows (new tuples added), then we drill down. If it underflows (tuples deleted), then drill up. This process is repeated until a valid node is reached when the sample is replaced by a randomly chosen tuple from its results. This estimator is still unbiased and can utilize the queries saved by reusing the signatures for obtaining additional samples in each round. Note that when database does not change between successive rounds, the queries issued by REISSUE-ESTIMATOR are always a subset of those issued by RESTART-ESTIMATOR resulting in substantial query savings. In the worst case, when the database is completely recreated in each round, REISSUE-ESTIMATOR *might* end up performing worse than RESTART-ESTIMATOR. In practice, REISSUE-ESTIMATE often results in query savings that could be used to retrieve new samples in that round. The new samples results in reduction of variance (as variance of the aggregate is inversely proportional to number of samples).

This idea could be extended further to design an even more sophisticated RS-ESTIMATOR[3] that distributes the query budget available for each round into two parts: one for reissuing (i.e., updating) drill downs

from previous rounds, and the other for initiating new drill downs. Intuitively, the distribution must be computed based on the amount of changes in the aggregate to be estimated. [3] describes the relevant optimization information that could be used for the distribution of query cost that are usually approximated via pilot samples. Intuitively, when the database undergoes little change, RS-ESTIMATOR mostly conducts new drill downs - leading to a lower estimation error than REISSUE and when the database changes drastically, RS-ESTIMATOR will be automatically reduced to REISSUE-ESTIMATOR[3].

6 Component TIMBR

In this section, we describe the TIMBR component of HYDRA which is responsible for three tasks: (1) translating a search query supported by a hidden database to the corresponding HTTP request that can be submitted to the website; (2) interpreting a webpage displaying the returned query answer to extract the corresponding tuple values; and (3) the proper scheduling of queries (i.e., HTTP requests submitted to websites), especially when there is a large number of concurrent data analytics tasks running in the HYDRA system. The following three subsections depict our design of the three components, respectively. As we mentioned in the introduction, it is important to understand that our goal here is not to develop new research results for the vast field of information extraction, but to devise a simple solution that fits our goal of the HYDRA system - i.e., a tool that a data analyst with substantial knowledge of the underlying data (as otherwise he/she would not be able to specify the aggregate queries anyway) can quickly deploy over a form-like hidden database. One can see in the following discussions how our design of TIMBR leverages two special properties of our system - the auxiliary input from a human data analyst; and the (bootstrapped) feedback from the SAMPLE-GEN and SAMPLE-EVAL components.

6.1 Input Interface Modeling

We start with the first task of TIMBR, input interface modeling, i.e., how to translate a search query supported by a hidden database to a corresponding HTTP request that can be transmitted to the web server of the hidden database. Before describing our design, we first outline two main technical challenges facing the modeling of an input interface: First is the increasingly complex nature of the (output) HTTP request. Traditionally (e.g., era before 2008), form-like search interfaces were usually implemented as simple HTML forms, with input values (i.e., search predicates) transmitted as HTTP GET or POST parameters. For these interfaces, the only thing our input-interface modeling component needs to identify is a URL plus a specification of how each attribute (i.e., predicate) is mapped to a GET or POST parameter. Here is an example:

URL: `http://www.autodb.com/`
Parameter: Make = p1 (GET), Model = p2 (GET)
Composed HTTP GET request: `http://www.autodb.com/search?p1=ford&p2=F150`

Nonetheless, more recently designed websites often use complex Ajax (i.e., asynchronous JavaScript) requests to transmit search queries to the web server - leading to sometimes not one, but multiple HTTP requests and responses that are interconnected with complex logic. For example, Figure 5 demonstrates an example with disney.com, which uses one Ajax request to retrieve an authentication token before using it in a second Ajax request to obtain the real query answer.

One can see that, for websites with such complex designs, it becomes impossible to properly model its input interface without parsing through the corresponding JavaScript source code. The second challenge stems from the first one, and is also an implication of the more powerful and flexible HTML5/JavaScript designs that are rapidly gaining popularity. Note that if the first challenge did not exist - i.e., if the output of interface modeling were simply an HTTP GET or POST request as in the above example - then we would not care about the design of the input (i.e., search interface) webpage at all. The reason is that, whatever the design might be, we only need to capture the submission URL and figure out the mapping between attributes and

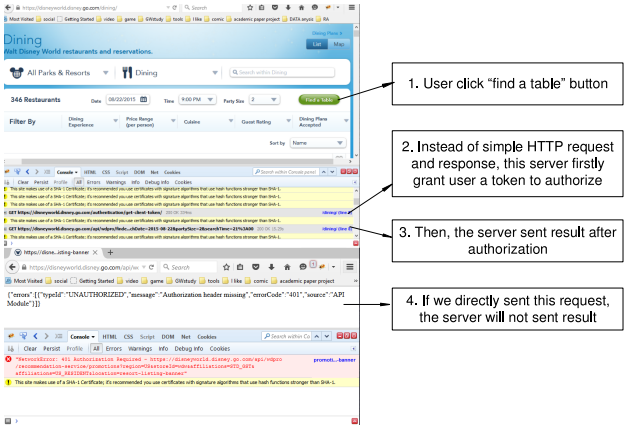


Figure 5: Example of multiple HTTP requests/responses

GET/POST parameters. However, given the presence of the first challenge and the necessity of “understanding” the JavaScript code required for query submission, it becomes necessary for us to identify the interface elements (and correspondingly, the JavaScript variables) capturing the user-specified values for each attribute.

It is from this requirement that the second challenge arises. Here the difficulty stems from the complex design of input controls e.g., textboxes or dropdown menus for specifying search predicates. Take textbox as an example. Traditionally, it is often implemented as a simple <input> tag that can be easily identified and mapped to an attribute according to minimal user interactions. Now, however, many websites such as Walmart implement textboxes in a completely customized manner, e.g., as a simple that responds to mouse clicks and/or keyboard events using custom-designed functions. Once again, this makes JavaScript parsing a prerequisite for input interface modeling.

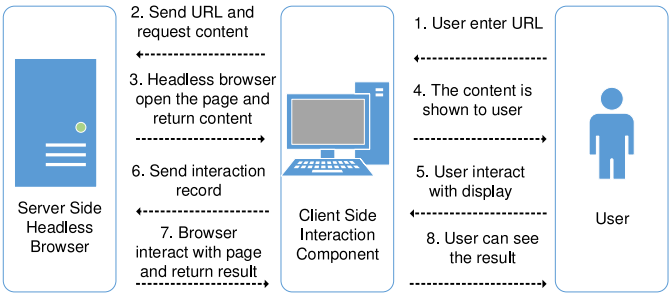


Figure 6: Process of Timbr

In light of the two challenges, our main technique for input interface modeling is not to parse the HTML and JavaScript code in our system, but to instead leverage an existing headless browser, e.g., PhantomJS, which runs on our HYDRA server. Figure 6 demonstrates the process of constructing an input-interface model. We start by doing two things: starting a headless browser at the backend and displaying the target website (i.e., the hidden database) as part of the HYDRA interface at the frontend. Then, we ask the user to interact with the displayed website to specify the input search conditions. The exact user interactions with the display (i.e., mouse clicks, keyboard events, etc.) are captured and transmitted to our backend headless browser, which repeats these interactions and (in the future) adjusts them - e.g., by changing the value specified for an attribute to form other search queries.

One can see here a key premise of our design: to convert the problem of modeling “how the input webpage interacts with the hidden database server” - a tedious task that requires complex HTML/JavaScript parsing -

to simply model "how the user interacts with the input webpage" - a much simpler task given its (general) transparency to the website Ajax design, yet can solve the same problem with the help of a headless browser running on HYDRA server. Of course, our approach calls for more user inputs than what is required by the fully automated interface modelers proposed in the literature (like [11, 12]) - e.g., instead of using techniques such as image pattern recognition [13] to automatically identify attribute controls, the mapping between an attribute and its corresponding webpage control (i.e., interaction) is defined by the human user in our TIMBR component. Nonetheless, we note that, as discussed in the introduction, our solution here is simpler and fits our goal of developing a fast-prototyping module for data analysts who already have substantial knowledge of the domain of the underlying data.

The detailed implementation calls for the proper treatment of many subtle issues. Some examples include how to properly measure the delay required between two operations (e.g., when the first operation activates an input control and the second one specifies the input value), how to detect and identify the correlation between different operations (e.g., only when car make is selected will a drop-down menu of car model appear), etc. Due to the space limitation here, we refer readers to our system documentation for details of these design issues [14].

6.2 Output Modeling

Compared with the input side, output modeling is a simpler task as the goal here is mainly to extract information from the returned result page, with only a few user interactions to capture. In the following discussions, we first describe our techniques for extracting tuples and attributes from the webpage, and then briefly discuss the interactions that need to be captured.

Extracting Tuples and Attributes: Consider the Document Object Model (DOM) tree of the result webpage. The goal here is to identify the group of subtrees corresponding to the tuples being returned and (at a finer granularity) the position of each attribute within a subtree. Ideally, all tuples should be mapped to the same level on the DOM tree, making the identification a straightforward process - so long as the user specify (by clicking on the HTML element corresponding to) the subtree of one example tuple, the remaining job is to simply find all other subtrees at the same level.

The practical cases are often more complex. For example, some websites place a subset of tuples (often those "featured" ones) into a special element, making them one level lower than the rest of tuples. In TIMBR, we address these challenges with two main ideas, namely human- and data-assisted tuple specification, respectively: the human-assisted approach asks a user to specify two tuples (ideally one at the beginning and the other at the end), in order to find the correct common ancestor HTML element for all tuples. Then, based on auxiliary information such as common CSS classes, the identification of tuple subtrees under this common ancestor becomes straightforward. The data-assisted approach is mainly used when such auxiliary information is not available. In this case, we mark as the "signature" of each subtree the positions and types of its elements that are identifiable using pre-known attribute domains (e.g., ZIP codes in US are five-digit combinations, US phone numbers are of the format 3 digits + 3 digits + 4 digits). Such a signature is then used to properly identify the subtrees under the common ancestor that are corresponding to the returned tuples.

Interaction Modeling: The interactions we need capture from the output interface are those "next page" or "load more" operations - they retrieve additional tuples that cannot be returned on the current page. Given the popularity of "infinite scroll" - i.e., Ajax-based loading of additional tuples without refreshing URL - in recently designed websites, we once again use the headless browser described in the input modeling part to capture the interaction and trigger it to retrieve additional tuples. A challenge with this design, however, is duplicate prevention. Specifically, some websites (e.g., *http://m.bloomingdales.com*) make the "next page" button available even when the returned results have already been exhausted. When a user clicks on the next page button, the same tuples will be displayed, leading to duplicate results. Note that we cannot simply compute a hash of the entire webpage HTML for duplicate detection, as peripheral contents (e.g., advertisements) on the page might change

with each load. Thus, in TIMBR we first extract tuple and attribute values from each page, and then compare the extracted values against the history to identify any duplications.

7 Related Work

Traditional Database Sampling and Approximate Query Processing: Sampling in traditional databases have a number of applications such as selectivity estimation, query optimization, approximate query processing etc. [15] proposed some of the earliest methods for enabling sampling in a database that has been extended by subsequent work. Most uniform sampling approaches suffer from low selectivity problem which is often addressed by biased samples. A number of weighted sampling techniques [16, 17] has been proposed. In particular, [17] exploits workload information to continuously tune the samples. Additional techniques for overcoming limitations of sampling such as outlier indexing[18], stratified sampling[19], dynamic sampling selection[20] has also been proposed. [21] proposed an online sampling technique with a time-accuracy tradeoff while [22] extended it by introducing novel methods of joining database tables.

There has been extensive work on approximate aggregate query processing over databases using sampling based techniques [23, 24, 25] and non sampling based techniques such as histograms [26] and wavelets [27]. Please refer to [28] for a survey. A common technique is to build a synopsis data structure that is a concise yet reasonably accurate representation of the database which can then be used to perform aggregate estimation. Maintenance of statistical aggregates in the presence of database updates have been considered in [29, 30, 31].

Hidden Database Crawling, Sampling and Analytics: The problem of crawling a hidden structured database has been extensively studied [32, 33, 34, 35, 36, 1]. The early works focussed on identifying and formulating effective queries over HTML forms so as to retrieve as many tuples as possible. In particular, [35] proposed an highly automatic and scalable mechanism for crawling hidden web databases by choosing an effective subset of search space of all possible input combinations that returns relevant tuples. [1] proposed optimal algorithms to completely crawl a hidden database with minimum queries and also provided theoretical results for the upper and lower bounds of query cost required for crawling.

There has been a number of prior work in performing sampling and aggregate estimation over hidden databases. [4, 5, 10] describe efficient techniques to obtain random samples from hidden web databases that can then be utilized to perform aggregate estimation. [2] provided an unbiased estimator for COUNT and SUM aggregates for databases with form based interfaces. [37] proposed an adaptive sampling algorithm for aggregation query processing over databases with hierarchical structure. Efficient algorithms for aggregate estimation over dynamic databases was proposed in [3]. There has been a number of prior work on enabling analytics over structured hidden databases such as generating content summaries[38, 39, 40], processing top-k queries[41], frequent itemset mining[42, 43], differential rule mining[44]. A number of techniques for variance reduction for analytics such as weight adjustment[2], divide-and-conquer[2], stratified sampling[42] and adaptive sampling[43, 45] has also been proposed.

Search Engine Sampling and Analytics: A number of prior work have considered crawling a search engine's corpus such as [46, 47, 48, 49] where the key objective was discovering legitimate query keywords. Most existing techniques over a search engine's corpus require prior knowledge of a query pool . Among them, [50] presented a method to measure the relative sizes between two search engines' corpora. [51, 52] achieved significant improvement on quality of samples and aggregate estimation. [53] introduced the concept of designated query, that allows one to completely remove sampling bias. Sampling databases through online suggestions was discussed in [54]. The key issue with requiring a query pool is that constructing a "good" query pool requires detailed knowledge of the database such as its size, topic, popular terms etc. There exist a handful of papers that mine a search engine's corpus without a query pool [51, 55] that operate by constructing a document graph (not fully materialized) and performing an on-the-fly random walks over them.

Information Extraction: There has been extensive prior work on information integration and extraction over hidden databases - see related tutorials [56, 57]. Parsing and understanding web query interfaces has been extensively studied (e.g., [13, 58]). The mapping of attributes across different web interfaces has been studied (e.g., [59]) while techniques for integrating query interfaces for multiple web databases can be found in [60, 61].

8 Final Remarks and Future Work

In this paper, we provided an overview of System HYDRA which enables fast sampling and data analytics over a hidden web database that provides nothing but a form-like web search interface as its only access channel. Specifically, we discussed three key components of HYDRA: SAMPLE-GEN which produces samples according to a given sampling distribution, SAMPLE-EVAL which generates estimations for a given aggregate query, and TIMBR which enables the fast and easy construction of a wrapper that translates a supported search query to HTTP requests and retrieves top-k query answers from HTTP responses.

It is our belief and vision that HYDRA enables a wide range of future research on hidden web databases. Within the database community, we think the research of HYDRA raises a fundamental question: exactly what kind of information can be inferred from a search-query-only, top-k-limited, access interface? HYDRA provides part of the answer by demonstrating that samples and aggregate query answers can be inferred from such an interface. But more research is needed to determine what other information can be inferred - and for other types of access interfaces/channels as well. Outside of the database community, we believe HYDRA provides a platform that enables inter-disciplinary studies with domain experts from other fields such as economy, sociology, etc., to investigate and understand the vast amount of data in real-world hidden web databases. It is our hope that HYDRA marks the start of an era that witnesses the effective and wide-spread usage of valuable analytics information in the deep web.

References

[1] C. Sheng, N. Zhang, Y. Tao, and X. Jin, "Optimal algorithms for crawling a hidden database in the web," *Proceedings of the VLDB Endowment*, vol. 5, no. 11, pp. 1112–1123, 2012.

[2] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das, "Unbiased estimation of size and other aggregates over hidden web databases," in *SIGMOD*, 2010.

[3] W. Liu, S. Thirumuruganathan, N. Zhang, and G. Das, "Aggregate estimation over dynamic hidden web databases," *Proceedings of the VLDB Endowment*, vol. 7, no. 12, pp. 1107–1118, 2014.

[4] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.

[5] A. Dasgupta, N. Zhang, and G. Das, "Leveraging count information in sampling hidden databases," in *ICDE*, 2009.

[6] W. Liu, M. F. Rahman, S. Thirumuruganathan, N. Zhang, and G. Das, "Aggregate estimations over location based services," *PVLDB*, vol. 8, no. 12, pp. 1334–1345, 2015.

[7] M. De Berg, M. Van Kreveld, M. Overmars, and O. C. Schwarzkopf, *Computational geometry*. Springer, 2000.

[8] N. Zhang and G. Das, "Exploration of deep web repositories," in *Tutorial, VLDB*, 2011.

[9] W. G. Cochran, *Sampling techniques*. John Wiley & Sons, 2007.

[10] A. Dasgupta, N. Zhang, and G. Das, "Turbo-charging hidden database samplers with overflowing queries and skew reduction," in *EDBT*, 2010.

[11] V. Crescenzi, G. Mecca, P. Merialdo *et al.*, "Roadrunner: Towards automatic data extraction from large web sites," in *VLDB*, vol. 1, 2001, pp. 109–118.

[12] G. Gkotsis, K. Stepanyan, A. I. Cristea, and M. Joy, "Entropy-based automated wrapper generation for weblog data extraction," *World Wide Web*, vol. 17, no. 4, pp. 827–846, 2014.

[13] E. Dragut, T. Kabisch, C. Yu, and U. Leser, "A hierarchical approach to model web query interfaces for web source integration," in *VLDB*, 2009.

[14] "TIMBR system documentation," `http://timbr.me/documents/guide.pdf`.

[15] F. Olken, "Random sampling from databases," Ph.D. dissertation, University of California at Berkeley, 1993.

[16] S. Acharya, P. B. Gibbons, and V. Poosala, "Congressional samples for approximate answering of group-by queries," in *ACM SIGMOD Record*, vol. 29, no. 2. ACM, 2000, pp. 487–498.

[17] V. Ganti, M.-L. Lee, and R. Ramakrishnan, "Icicles: Self-tuning samples for approximate query answering." in *VLDB*, vol. 176. Citeseer, 2000, p. 187.

[18] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya, "Overcoming limitations of sampling for aggregation queries," in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 534–542.

[19] S. Chaudhuri, G. Das, and V. Narasayya, "A robust, optimization-based approach for approximate answering of aggregate queries," in *ACM SIGMOD Record*, vol. 30, no. 2. ACM, 2001, pp. 295–306.

[20] B. Babcock, S. Chaudhuri, and G. Das, "Dynamic sample selection for approximate query processing," in *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. ACM, 2003, pp. 539–550.

[21] J. M. Hellerstein, P. J. Haas, and H. J. Wang, "Online aggregation," *ACM SIGMOD Record*, vol. 26, no. 2, pp. 171–182, 1997.

[22] P. J. Haas and J. M. Hellerstein, "Ripple joins for online aggregation," in *ACM SIGMOD Record*, vol. 28, no. 2. ACM, 1999, pp. 287–298.

[23] S. Chaudhuri, G. Das, and V. R. Narasayya, "A robust, optimization-based approach for approximate answering of aggregate queries." in *SIGMOD*, 2001.

[24] S. Chaudhuri, G. Das, and V. R. Narasayya, "Optimized stratified sampling for approximate query processing." *ACM Trans. Database Syst.*, vol. 32, no. 2, p. 9, 2007.

[25] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. R. Narasayya, "Overcoming limitations of sampling for aggregation queries," in *ICDE*, 2001.

[26] V. Poosala and V. Ganti, "Fast approximate query answering using precomputed statistics." in *ICDE*, 1999, p. 252.

[27] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim, "Approximate query processing using wavelets," in *The VLDB Journal*, 2000, pp. 111–122.

[28] M. N. Garofalakis and P. B. Gibbons, "Approximate query processing: Taming the terabytes," in *VLDB*, 2001.

[29] P. B. Gibbons and Y. Matias, "Synopsis data structures for massive data sets," in *SODA*, 1999, pp. 909–910.

[30] J. Gehrke, F. Korn, and D. Srivastava, "On computing correlated aggregates over continual data streams," in *SIGMOD*, 2001, pp. 13–24.

[31] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams," in *SIGMOD*, 2002.

[32] M. Álvarez, J. Raposo, A. Pan, F. Cacheda, F. Bellas, and V. Carneiro, "Crawling the content hidden behind web forms," *Computational Science and Its Applications–ICCSA 2007*, pp. 322–333, 2007.

[33] A. Calì and D. Martinenghi, "Querying the deep web," in *Proceedings of the 13th International Conference on Extending Database Technology*. ACM, 2010, pp. 724–727.

[34] S. W. Liddle, D. W. Embley, D. T. Scott, and S. H. Yau, "Extracting data behind web forms," in *Advanced Conceptual Modeling Techniques*. Springer, 2003, pp. 402–413.

[35] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, "Google's deep web crawl," *Proceedings of the VLDB Endowment*, vol. 1, no. 2, pp. 1241–1252, 2008.

[36] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," in *Proceedings of the 27th International Conference on Very Large Data Bases*, ser. VLDB '01. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001, pp. 129–138.

[37] F. N. Afrati, P. V. Lekeas, and C. Li, "Adaptive-sampling algorithms for answering aggregation queries on web sites," *DKE*, vol. 64, no. 2, pp. 462–490, 2008.

[38] J. Callan and M. Connell, "Query-based sampling of text databases," *ACM TOIS*, vol. 19, no. 2, pp. 97–130, 2001.

[39] P. Ipeirotis and L. Gravano, "Distributed search over the hidden web: Hierarchical database sampling and selection," in *VLDB*, 2002.

[40] Y.-L. Hedley, M. Younas, A. E. James, and M. Sanderson, "Sampling, information extraction and summarisation of hidden web databases," *DKE*, vol. 59, no. 2, pp. 213–230, 2006.

[41] N. Bruno, L. Gravano, and A. Marian, "Evaluating top-k queries over web-accessible databases," in *International Conference on Data Engineering*, 2002.

[42] T. Liu, F. Wang, and G. Agrawal, "Stratified sampling for data mining on the deep web," *Frontiers of Computer Science*, vol. 6, no. 2, pp. 179–196, 2012.

[43] F. Wang and G. Agrawal, "Effective and efficient sampling methods for deep web aggregation queries," in *Proceedings of the 14th International Conference on Extending Database Technology*. ACM, 2011, pp. 425–436.

[44] T. Liu and G. Agrawal, "Active learning based frequent itemset mining over the deep web," in *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*. IEEE, 2011, pp. 219–230.

[45] T. Liu, F. Wang, and G. Agrawal, "Stratified sampling for data mining on the deep web," *Frontiers of Computer Science*, vol. 6, no. 2, pp. 179–196, 2012.

[46] E. Agichtein, P. G. Ipeirotis, and L. Gravano, "Modeling query-based access to text databases," in *WebDB*, 2003.

[47] A. Ntoulas, P. Zerfos, and J. Cho, "Downloading textual hidden web content through keyword queries," in *JCDL*, 2005.

[48] L. Barbosa and J. Freire, "Siphoning hidden-web data through keyword-based interfaces." in *SBBD*, 2004, pp. 309–321.

[49] K. Vieira, L. Barbosa, J. Freire, and A. Silva, "Siphon++: a hidden-webcrawler for keyword-based interfaces," in *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM, 2008, pp. 1361–1362.

[50] K. Bharat and A. Broder, "A technique for measuring the relative size and overlap of public web search engines," in *WWW*, 1998.

[51] Z. Bar-Yossef and M. Gurevich, "Random sampling from a search engine's corpus," *Journal of the ACM*, vol. 55, no. 5, 2008.

[52] Z. Bar-Yossef and M. Gurevich, "Efficient search engine measurements," in *WWW*, 2007.

[53] M. Zhang, N. Zhang, and G. Das, "Mining a search engine's corpus: efficient yet unbiased sampling and aggregate estimation," in *SIGMOD*, 2011, pp. 793–804.

[54] Z. Bar-Yossef and M. Gurevich, "Mining search engine query logs via suggestion sampling," in *VLDB*, 2008.

[55] M. Zhang, N. Zhang, and G. Das, "Mining a search engine's corpus without a query pool," in *Proceedings of the 22nd ACM international conference on Conference on information & knowledge management*. ACM, 2013, pp. 29–38.

[56] K. Chang and J. Cho, "Accessing the web: From search to integration," in *Tutorial, SIGMOD*, 2006.

[57] A. Doan, R. Ramakrishnan, and S. Vaithyanathan, "Managing information extraction," in *Tutorial, SIGMOD*, 2006.

[58] Z. Zhang, B. He, and K. Chang, "Understanding web query interfaces: best-effort parsing with hidden syntax," in *SIGMOD*, 2004.

[59] B. He, K. Chang, and J. Han, "Discovering complex matchings across web query interfaces: A correlation mining approach," in *KDD*, 2004.

[60] E. Dragut, C. Yu, and W. Meng, "Meaningful labeling of integrated query interfaces," in *VLDB*, 2006.

[61] B. He and K. Chang, "Statistical schema matching across web query interfaces," in *SIGMOD*, 2003.

Approximate Geometric Query Tracking over Distributed Streams

Minos Garofalakis
School of Electronic and Computer Engineering
Technical University of Crete
minos@softnet.tuc.gr

Abstract

Effective Big Data analytics pose several difficult challenges for modern data management architectures. One key such challenge arises from the naturally streaming nature of big data, which mandates efficient algorithms for querying and analyzing massive, continuous data streams (that is, data that is seen only once and in a fixed order) with limited memory and CPU-time resources. Such streams arise naturally in emerging large-scale event monitoring applications; for instance, network-operations monitoring in large ISPs, where usage information from numerous sites needs to be continuously collected and analyzed for interesting trends. In addition to memory- and time-efficiency concerns, the inherently distributed nature of such applications also raises important communication-efficiency issues, making it critical to carefully optimize the use of the underlying network infrastructure. In this paper, we provide a brief introduction to the distributed data streaming model and the Geometric Method (GM), a generic technique for effectively tracking complex queries over massive distributed streams. We also discuss several recently-proposed extensions to the basic GM framework, such as the combination with stream-sketching tools and local prediction models, as well as more recent developments leading to a more general theory of Safe Zones and interesting connections to convex Euclidean geometry. Finally, we outline various challenging directions for future research in this area.

1 Introduction

Traditional data-management systems are typically built on a *pull-based paradigm*, where users issue one-shot queries to static data sets residing on disk, and the system processes these queries and returns their results. For several emerging application domains, however, data arrives and needs to be processed on a continuous (24×7) basis, without the benefit of several passes over a static, persistent data image. These *continuous data streams* arise naturally in new large-scale event monitoring applications, that require the ability to efficiently process continuous, high-volume streams of data in real time. Such monitoring systems are routinely employed, for instance, in the network installations of large Telecom and Internet service providers where detailed usage information (Call-Detail-Records (CDRs), SNMP/RMON packet-flow data, etc.) from different parts of the underlying network needs to be continuously collected and analyzed for interesting trends. Other examples

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

include real-time analysis tools for financial data streams, and event and operations monitoring applications for enterprise clouds and data centers. As both the scale of today's networked systems, and the volumes and rates of the associated data streams continue to increase with no bound in sight, algorithms and tools for effectively analyzing them are becoming an important research mandate.

Large-scale stream processing applications rely on *continuous*, event-driven monitoring, that is, real-time tracking of measurements and events, rather than one-shot answers to sporadic queries. Furthermore, the vast majority of these applications are inherently *distributed*, with several remote monitor sites observing their local, high-speed data streams and exchanging information through a communication network. This distribution of the data naturally implies critical communication constraints that typically prohibit centralizing all the streaming data, due to either the huge volume of the data (e.g., in IP-network monitoring, where the massive amounts of collected utilization and traffic information can overwhelm the production IP network [13]), or power and bandwidth restrictions (e.g., in wireless sensornets, where communication is the key determinant of sensor battery life [28]). Finally, an important requirement of large-scale event monitoring is the effective support for tracking complex, *holistic queries* that provide a global view of the data by combining and correlating information across the collection of remote monitor sites. For instance, tracking aggregates over the result of a distributed join (the "workhorse" operator for combining tables in relational databases) can provide unique, real-time insights into the workings of a large-scale distributed system, including system-wide correlations and potential anomalies [7]. Monitoring the precise value of such holistic queries without continuously centralizing all the data seems hopeless; luckily, when tracking statistical behavior and patters in large scale systems, *approximate answers* (with reasonable approximation error guarantees) are often sufficient. This often allows algorithms to effectively tradeoff efficiency with approximation quality (e.g., using sketch-based stream approximations [7]).

Given the prohibitive cost of data centralization, it is clear that realizing sophisticated, large-scale distributed data-stream analysis tools must rely on novel algorithmic paradigms for processing local streams of data *in situ* (i.e., locally at the sites where the data is observed). This, of course, implies the need for intelligently decomposing a (possibly complex) global data-analysis and monitoring query into a collection of "safe" local queries that can be tracked independently at each site (without communication), while guaranteeing correctness for the global monitoring operation. This decomposition process can enable truly distributed, event-driven processing of real-time streaming data, using a *push-based paradigm*, where sites monitor their local queries and communicate only when some local query constraints are violated [7, 34]. Nevertheless, effectively decomposing a complex, holistic query over the global collections of streams into such local constraints is far from straightforward, especially in the case of *non-linear* queries (e.g., norms or joins) [34].

The bulk of early work on data-stream processing has focused on developing space-efficient, one-pass algorithms for performing a wide range of *centralized computations* on massive data streams; examples include computing quantiles [22], estimating distinct values [20], and set-expression cardinalities [16], counting frequent elements (i.e., "heavy hitters") [5, 11, 29], approximating large Haar-wavelet coefficients [10], and estimating join sizes and stream norms [1, 2, 15]. Monitoring *distributed* data streams has attracted substantial research interest in recent years [6, 31], with early work focusing on the monitoring of *single values*, and building appropriate models and filters to avoid propagating updates if these are insignificant compared to the value of simple *linear* aggregates (e.g., to the `SUM` of the distributed values). For instance, [32] proposes a scheme based on "adaptive filters" — that is, bounds around the value of distributed variables, which shrink or grow in response to relative stability or variability, while ensuring that the total uncertainty in the bounds is at most a user-specified bound. Still, in the case of linear aggregate functions, deriving local filter bounds based on a global monitoring condition is rather straightforward, with the key issue being how to intelligently distribute the available aggregate "slack" across all sites [3, 9, 24].

In this paper, we focus on recently-developed algorithmic tools for effectively tracking a broad class of complex queries over massive, distributed data streams. We start by describing the key elements of a generic *distributed stream-processing model* and define a broad class of distributed query-tracking problems addressed by our techniques. We then give an overview of the *Geometric Method (GM)* [34, 25] for distributed threshold

monitoring that lies at the core of our distributed query-tracking methodology, and briefly discuss recent extensions to the basic GM framework that incorporate stream sketches [17] and local prediction models [18, 19]. We also summarize recent developments leading to a more general theory of *Safe Zones* for geometric monitoring and interesting connections to convex Euclidean geometry [27]. Finally, we conclude with a brief discussion of new research directions in this space.

2 Distributed Data Streaming and the Geometric Method

Data-Stream Processing. Recent years have witnessed an increasing interest in designing data-processing algorithms that work over continuous data streams, i.e., algorithms that provide results to user queries while looking at the relevant data items *only once and in a fixed order* (determined by the stream-arrival pattern). Data-stream processing turns the paradigm of conventional database systems on its head: Databases typically have to deal with a stream of queries over a static, bounded data set; instead, a stream processing engine has to effectively process a static set of queries over continuous streams of data. Such stream queries are typically *continuous*, implying the need for continuous, real-time monitoring of the query answer over the changing stream.

Formally, a data stream can be modeled as a *massive, dynamic, one-dimensional vector* $\boldsymbol{v}[1 \ldots N]$ that, at any point in time, captures the current state of the stream. Note that this is a very generic, powerful model — for instance, in the case of streams of relational tuples (rendering a dynamic relational table), this vector $\boldsymbol{v}$ is essentially the (dynamic) frequency distribution vector of the underlying relational table whose values capture the counts of different tuples (i.e., attribute value combinations) in the relation. (Multi-attribute relational tables can naturally be handled in this abstract model by simply "unfolding" the corresponding multi-dimensional frequency distribution on one vector dimension using standard techniques, e.g., row- or column-major). As an example, in the case of IP routers monitoring the number of `TCP` connections and `UDP` packets exchanged between source and destination IP addresses, the stream vector $\boldsymbol{v}$ has 2×2^{64} entries capturing the up-to-date frequencies for specific (source, destination) pairs observed in `TCP` connections and `UDP` packets routed through router j. (For instance, the first (last) 2^{64} entries of $\boldsymbol{v}$ could be used for TCP-connection (respectively, UDP-packet) frequencies.) The size N of the stream vector $\boldsymbol{v}$ vector is defined as the product of the attribute domain size(s) which can easily grow very large. [1] The dynamic vector $\boldsymbol{v}$ is rendered through a continuous stream of updates, where each update effectively modifies values in $\boldsymbol{v}$ — the nature of these update operations gives rise to different data streaming models, such as *time-series*, *cash-register*, and *turnstile* streams [30].

Data-stream processing algorithms aim to compute functions (or, queries) on the stream vector $\boldsymbol{v}$ at different points during the lifetime of the stream (continuous or ad-hoc). Since N can be very large, the typical requirement here is that these algorithms work in *small space* (i.e., the state maintained by the algorithm) and *small time* (i.e., the processing time per update), where "small" is understood to mean a quantity significantly smaller than $\Theta(N)$ (typically, poly-logarithmic in N). Several such stream-processing algorithms are known for various data-analysis queries [1, 2, 5, 10, 11, 15, 16, 20, 22, 29].

Distributed Data Streaming. The naturally distributed nature of large-scale event-monitoring applications (such as the ones mentioned earlier) implies one additional level of complexity, in the sense that there is no centralized observation point for the dynamic stream vector $\boldsymbol{v}$; instead, $\boldsymbol{v}$ is distributed across several sites. More specifically, we consider a distributed computing environment, comprising a collection of k *remote sites* and a designated *coordinator site*. Streams of data updates arrive continuously at remote sites, while the coordinator site is responsible for generating approximate answers to (possibly, continuous) user queries posed over the collection of remotely-observed streams (across all sites). Following earlier work in the area [3, 7, 9, 14, 32], our distributed stream-processing model does not explicitly allow direct communication between remote sites;

[1] Note that streaming algorithms typically do not require a priori knowledge of N.

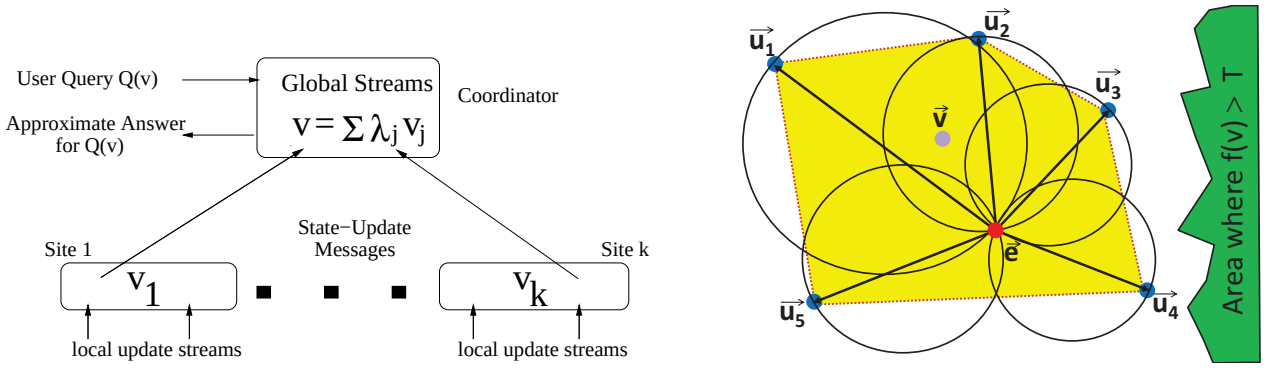


Figure 1: (a) Distributed stream processing architecture. (b) Geometric Method: Estimate vector $\vec{e}$, drift vectors $\boldsymbol{u}_j$, convex hull enclosing current $\boldsymbol{v}$ (dotted outline), and bounding balls $B(\boldsymbol{e} + \frac{1}{2}\Delta\boldsymbol{v}_j, \frac{1}{2}\|\Delta\boldsymbol{v}_j\|)$.

instead, as illustrated in Figure 1(a), a remote site exchanges messages only with the coordinator, providing it with state information on its (locally-observed) streams. [2] Note that such a hierarchical processing model is, in fact, representative of a large class of applications, including network monitoring where a central Network Operations Center (NOC) is responsible for processing network traffic statistics (e.g., link bandwidth utilization, IP source-destination byte counts) collected at switches, routers, and/or Element Management Systems (EMSs) distributed across the network.

Each remote site $j \in \{1, \ldots, k\}$ observes (possibly, several) local update streams that incrementally render a *local stream vector* $\boldsymbol{v}_j$ capturing the current local state of the observed stream(s) at site j. All local stream vectors $\boldsymbol{v}_j$ in our distributed streaming architecture change dynamically over time — when necessary, we make this dependence explicit, using $\boldsymbol{v}_j(t)$ to denote the state of the vector at time t (assuming a consistent notion of "global time" in our distributed system). The unqualified notation $\boldsymbol{v}_j$ typically refers to the *current* state of the local stream vector.

We define the *global stream vector* $\boldsymbol{v}$ of our distributed stream(s) as any *weighted average* (i.e., convex combination) of the local stream vectors $\{\boldsymbol{v}_j\}$; that is, $\boldsymbol{v} = \sum_{j=1}^{k} \lambda_j \boldsymbol{v}_j$, where $\sum_j \lambda_j = 1$ and $\lambda_j \geq 0$ for all j. (Again, to simplify notation, we typically omit the explicit dependence on time when referring to the current global vector.) Our focus is on the problem of effectively answering user queries (or, functions) over the global stream vector at the coordinator site. Rather than one-time query/function evaluation, we assume a continuous-querying environment which implies that the coordinator needs to *continuously maintain* (or, *track*) the answers to queries as the local update streams $\boldsymbol{v}_j$ evolve at individual remote sites. There are two defining characteristics of our problem setup that raise difficult algorithmic challenges for our query tracking problems:

• *The distributed nature and large volumes of local streaming data* raise important communication and space/-time efficiency concerns. Naïve schemes that accurately track query answers by forcing remote sites to ship every remote stream update to the coordinator are clearly impractical, since they can impose an inordinate burden on the underlying communication infrastructure (especially, for high-rate data streams and large numbers of remote sites). Furthermore, the voluminous nature of the local data streams implies that effective streaming tools are needed at the remote sites in order to manage the local stream vectors in sublinear space/time. Thus, a practical approach is to adopt the paradigm of continuous tracking of *approximate* query answers at the coordinator site with strong guarantees on the quality of the approximation. This allows schemes that can effectively trade-off space/time/communication efficiency and query-approximation accuracy in a precise, quantitative manner.

• *General, non-linear queries/functions* imply fundamental and difficult challenges for distributed monitoring.

[2] Of course, sites can always communicate with each other through the coordinator — this would only increase communication load by a factor of 2.

For the case of linear functions, a number of approaches have been proposed that rely on the key idea of allocating appropriate *"slacks"* to the remote sites based on their locally-observed function values (e.g., [3, 32, 24]). Unfortunately, it is not difficult to find examples of simple *non-linear* functions on one-dimensional data, where it is basically impossible to make any assumptions about the value of the global function based on the values observed locally at the sites [34]. This renders conventional slack-allocation schemes inapplicable in this more general setting.

The Geometric Method (GM). Sharfman et al. [34] consider the fundamental problem of *distributed threshold monitoring*; that is, determine whether $f(\boldsymbol{v}) < \tau$ or $f(\boldsymbol{v}) > \tau$, for a given (general) function $f()$ over the global stream vector and a fixed threshold τ. Their key idea is that, since it is generally impossible to connect the locally-observed values of $f()$ to the global value $f(\boldsymbol{v})$, one can employ geometric arguments to monitor the *domain* (rather than the range) of the monitored function $f()$. More specifically, assume that at any point in time, each site j has informed the coordinator of some prior state of its local vector $\boldsymbol{v}_j^p$; thus, the coordinator has an estimated global vector $\boldsymbol{e} = \boldsymbol{v}^p = \sum_{j=1}^{k} \lambda_j \boldsymbol{v}_j^p$. Clearly, the updates arriving at sites can cause the local vectors $\boldsymbol{v}_j$ to drift too far from their previously reported values $\boldsymbol{v}_j^p$, possibly leading to a violation of the τ threshold. Let $\Delta\boldsymbol{v}_j = \boldsymbol{v}_j - \boldsymbol{v}_j^p$ denote the local *delta vector* (due to updates) at site j, and let $\boldsymbol{u}_j = \boldsymbol{e} + \Delta\boldsymbol{v}_j$ be the *drift vector* from the previously reported estimate at site j. We can then express the current global stream vector $\boldsymbol{v}$ in terms of the drift vectors:

$$\boldsymbol{v} \;=\; \sum_{j=1}^{k} \lambda_j (\boldsymbol{v}_j^p + \Delta\boldsymbol{v}_j) \;=\; \boldsymbol{e} + \sum_{j=1}^{k} \lambda_j \Delta\boldsymbol{v}_j \;=\; \sum_{j=1}^{k} \lambda_j (\boldsymbol{e} + \Delta\boldsymbol{v}_j).$$

That is, the current global vector is a convex combination of drift vectors and, thus, guaranteed to lie somewhere within the convex hull of the delta vectors around $\boldsymbol{e}$. Figure 1(b) depicts an example in $d = 2$ dimensions. The current value of the global stream vector lies somewhere within the shaded convex-hull region; thus, as long as the convex hull does not overlap the inadmissible region (i.e., the region $\{\boldsymbol{v} \in \mathbb{R}^2 : f(\boldsymbol{v}) > \tau\}$ in Figure 1(b)), we can guarantee that the threshold has not been violated (i.e., $f(\boldsymbol{v}) \leq \tau$)).

The problem, of course, is that the $\Delta\boldsymbol{v}_j$'s are spread across the sites and, thus, the above condition cannot be checked locally. To transform the global condition into a local constraint, we place a d-dimensional *bounding ball* $B(\boldsymbol{c}, r)$ around each local delta vector, of radius $r = \frac{1}{2}\|\Delta\boldsymbol{v}_j\|$ and centered at $\boldsymbol{c} = \boldsymbol{e} + \frac{1}{2}\Delta\boldsymbol{v}_j$ (see Figure 1(b)). It can be shown that the union of all these balls completely covers the convex hull of the drift vectors [34]. This observation effectively reduces the problem of monitoring the global stream vector to the local problem of each remote site monitoring the ball around its local delta vector.

More specifically, given the monitored function $f()$ and threshold τ, we can partition the d-dimensional space into two sets $A = \{\boldsymbol{v} : f(\boldsymbol{v}) \leq \tau\}$ and $\overline{A} = \{\boldsymbol{v} : f(\boldsymbol{v}) > \tau\}$. (Note that these sets can be arbitrarily complex, e.g., they may comprise multiple disjoint regions of $\mathbb{R}^d$.) The basic protocol is now quite simple: Each site monitors its delta vector $\Delta\boldsymbol{v}_j$ and, with each update, checks whether its bounding ball $B(\boldsymbol{e} + \frac{1}{2}\Delta\boldsymbol{v}_j, \frac{1}{2}\|\Delta\boldsymbol{v}_j\|)$ is *monochromatic*, i.e., all points in the ball lie within the same region (A or $\overline{A}$). If this is not the case, we have a *local threshold violation*, and the site communicates its local $\Delta\boldsymbol{v}_j$ to the coordinator. The coordinator then initiates a *synchronization process* that typically tries to resolve the local violation by communicating with only a subset of the sites in order to "balance out" the violating $\Delta\boldsymbol{v}_j$ and ensure the monochromicity of all local bounding balls [34]. In the worst case, the delta vectors from all k sites are collected, leading to an accurate estimate of the current global stream vector, which is by definition monochromatic (since all bounding balls have 0 radius).

The power of the GM stems from the fact that it is essentially agnostic of the specific (global) function $f(\boldsymbol{v})$ being monitored.[3] Note that the function itself is only used at a remote site when checking the monochromicity

[3]The assumption that GM only monitors functions of the (weighted) average of local stream vectors is not really restrictive: Numerous complex functions can actually be expressed as functions of the average using simple tricks, such as adding additional dimensions to the stream vectors, e.g., [4].

of its local ball, which essentially boils down to solving a minimization/maximization problem for $f()$ within the area of that ball. This may, of course, be complex but it also enables the GM to effectively trade local computation for communication.

From Threshold Crossing to Approximate Query Tracking. Consider the task of monitoring (at the coordinator) the value of a function $f()$ over the global stream vector $\boldsymbol{v}$ to within θ relative error. (Our discussion here focuses on relative error – the case of monitoring to within bounded *absolute* error can be handled in a similar manner.) Since all the coordinator has is the estimated value of the global stream vector $\boldsymbol{e} = \boldsymbol{v}^p$ based on the most recent site updates $\boldsymbol{v}_j^p$, our monitoring protocol would have to guarantee that the estimated function value carries at most θ relative error compared to the up-to-date value $f(\boldsymbol{v}) = f(\boldsymbol{v}(t))$, that is $f(\boldsymbol{v}^p) \in (1 \pm \theta) f(\boldsymbol{v})$ [4], which is obviously equivalent to monitoring two threshold queries on $f(\boldsymbol{v})$:

$$f(\boldsymbol{v}) \geq \frac{f(\boldsymbol{v}^p)}{1+\theta} \quad \text{and} \quad f(\boldsymbol{v}) \leq \frac{f(\boldsymbol{v}^p)}{1-\theta}.$$

These are exactly the threshold conditions that our approximate function tracking protocols will need to monitor. Note that $f(\boldsymbol{v}^p)$ in the above expression is a constant (based on the latest communication of the coordinator with the remote sites). Similar threshold conditions can also be derived when the local/global values of $f()$ are only known approximately (e.g., using sketches [1, 2] or other streaming approximations) — the threshold conditions just need to account for the added approximation error [17].

3 Enhancing GM: Sketches and Prediction Models

In this section, we give an overview of more recent work on extending the GM with two key stream-processing tools, namely sketches and prediction models [17, 18, 19].

GM and AMS Sketches. Techniques based on small-space pseudo-random *sketch* summaries of the data have proved to be very effective tools for dealing with massive, rapid-rate data streams in centralized settings [8]. The key idea in such sketching techniques is to represent a streaming frequency vector $\boldsymbol{v}$ using a much smaller (typically, randomized) *sketch* vector (denoted by $\texttt{sk}(\boldsymbol{v})$) that (1) can be easily maintained as the updates incrementally rendering $\boldsymbol{v}$ are streaming by, and (2) provide probabilistic guarantees for the quality of the data approximation. The widely used AMS sketch (proposed by Alon, Matias, and Szegedy in their seminal paper [2]) defines the entries of the sketch vector as pseudo-random linear projections of $\boldsymbol{v}$ that can easily maintained over the stream of updates. AMS sketch estimators can effectively approximate *inner-product queries* $\boldsymbol{v} \cdot \boldsymbol{u} = \sum_i \boldsymbol{v}[i] \cdot \boldsymbol{u}[i]$ over streaming data vectors and tensors. Such inner products naturally map to several interesting query classes, including join and multi-join aggregates [15], range and quantile queries [21], heavy hitters and top-k queries [5], and approximate histogram and wavelet representations [10]. The AMS estimator function $f_{\text{AMS}}()$ computed over the sketch vectors of $\boldsymbol{v}$ and $\boldsymbol{u}$ is complex, and involves both averaging and median-selection over the components of the sketch-vector inner product [1, 2]. Formally, viewing each sketch vector as a two-dimensional $n \times m$ array (where $n = O(\frac{1}{\epsilon^2})$, $m = O(\log(1/\delta))$ and ϵ, $1-\delta$ denote desired bounds on error and probabilistic confidence (respectively)), the AMS estimator function is defined as:

$$f_{\text{AMS}}(\texttt{sk}(\boldsymbol{v}), \texttt{sk}(\boldsymbol{u})) \;=\; \underset{i=1,\ldots,m}{\texttt{median}} \left\{ \frac{1}{n} \sum_{l=1}^{n} \texttt{sk}(\boldsymbol{v})[l,i] \cdot \texttt{sk}(\boldsymbol{u})[l,i] \right\}, \tag{14}$$

and guarantees that, with probability $\geq 1-\delta$, $f_{\text{AMS}}(\texttt{sk}(\boldsymbol{v}), \texttt{sk}(\boldsymbol{u})) \in (\boldsymbol{v} \cdot \boldsymbol{u} \pm \epsilon \|\boldsymbol{v}\| \|\boldsymbol{u}\|)$ [1, 2].

Moving to the distributed streams setting, note that our discussion of the GM thus far has assumed that all remote sites maintain the *full stream vector* (i.e., employ $\Theta(N)$ space), which is often unrealistic for real-life data streams. In our recent work [17], we have proposed novel approximate query tracking protocols that

[4] Throughout, the notation $x \in (y \pm z)$ is equivalent to $|x-y| \leq |z|$.

exploit the combination of the GM and AMS sketch estimators. The AMS sketching idea offers an effective streaming dimensionality-reduction tool that significantly expands the scope of the original GM, allowing it to handle massive, high-dimensional distributed data streams in an efficient manner with approximation-quality guarantees. Furthermore, the linearity of AMS sketches implies that they can be trivially merged (by simple component-wise addition), making them particularly suitable to our distributed streams settings [7]. A key technical observation is that, by exploiting properties of the AMS estimator function, geometric monitoring can now take place in a *much lower-dimensional space*, allowing for communication-efficient monitoring. Another technical challenge that arises is how to effectively test the monochromicity of bounding balls in this lower-dimensional space with respect to threshold conditions involving the highly non-linear `median` operator in the AMS estimator $f_{\text{AMS}}()$ (Equation (14)). We have proposed a number of novel algorithmic techniques to address these technical challenges, starting from the easier cases of L_2-norm (i.e., self-join) and range queries, and then extending them to the case of general inner-product (i.e., binary-join) queries. Our experimental study with real-life data sets demonstrates the practical benefits of our approach, showing consistent gains of up to 35% in terms of total communication cost compared to state-of-the-art methods; furthermore, our techniques demonstrate even more impressive benefits (of over 100%) when focusing on the communication costs of data (i.e., sketch) shipping in the system.

GM and Prediction Models. In other recent work [18, 19], we have proposed a novel combination of the geometric method with *local prediction models* for describing the temporal evolution of local data streams. (The adoption of prediction models has already been proven beneficial in terms of bandwidth preservation in distributed settings [7].) We demonstrate that prediction models can be incorporated in a very natural way in the geometric method for tracking general, non-linear functions; furthermore, we show that the initial geometric monitoring method of Sharfman et al. [25, 34] is only a special case of our, more general, prediction-based geometric monitoring framework. Interestingly, the mere utilization of local predictions is not enough to guarantee lower communication overheads even when predictors are quite capable of describing local stream distributions. We establish a theoretically solid monitoring framework that incorporates conditions that can lead to fewer contacts with the coordinator. We also develop a number of mechanisms, along with extensive probabilistic models and analysis, that relax the previously introduced framework, base their function on simpler criteria, and yield significant communication benefits in practical scenarios.

4 Towards Convex Safe Zones

In followup work to the GM, Keren et al. [25] propose a simple, generic geometric monitoring strategy that can be formally shown to encompass the original GM scheme as a special case. Briefly, assuming we are monitoring the threshold condition $f(\boldsymbol{v}) \leq \tau$, the idea is to define a certain *convex subset* C of the admissible region $A = \{\boldsymbol{v} : f(\boldsymbol{v}) \leq \tau\}$ (i.e., a *convex admissible subset*), which is then used to define *Safe Zones (SZs)* for the local drift vectors: *Site j simply monitors the condition* $\boldsymbol{u}_j = \boldsymbol{e} + \Delta\boldsymbol{v}_j \in C$. The correctness of this generic monitoring scheme follows directly from the convexity of C, and our earlier observation that the global stream vector $\boldsymbol{v}$ always lies in the convex hull of $\boldsymbol{u}_j$, $j = 1, \ldots, k$: If $\boldsymbol{u}_j \in C$ for all nodes j then, by convexity, this convex hull (and, therefore $\boldsymbol{v}$) lies completely within C and, therefore, the admissible region (since $C \subseteq A$). (Note that the convexity of C plays a crucial role in the above correctness argument.)

While the convexity of C is needed for the *correctness* of the monitoring scheme, it is clear that the size of C plays a critical role in its *efficiency*: Obviously, a larger C implies fewer local violations and, thus, smaller communication/synchronization overheads. This, in turn, implies a fairly obvious dominance relationship over geometric distributed monitoring schemes: Given two geometric algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ (for the same distributed monitoring problem) that use the convex admissible subsets C_1 and C_2 (respectively), algorithm $\mathcal{A}_1$ is *provably superior* to $\mathcal{A}_2$ if $C_2 \subset C_1$. Note that, in the simple case of *linear* functions $f()$, the admissible region A itself is convex, and therefore one can choose $C = A$; however, for more complicated, non-linear functions, A is

non-convex and quite complex. Thus, finding a "large" convex subset of A is a crucial component of effective geometric monitoring.

Interestingly, the bounding ball constraints of the GM can also be cast in terms of a convex admissible subset (denoted by C_{GM}) that can be mathematically shown to be equivalent to the intersection of the (possibly, infinitely many) half-spaces defined by points at the boundary of the admissible region A [25]. Furthermore, as demonstrated in our recent work [27], while the GM can achieve good results and is generic (i.e., can be applied to any monitoring function), its performance can be far from optimal since its underlying SZ C_{GM} is often far too restrictive. In several practical scenarios, C_{GM} can be drastically improved by intersecting *much fewer half-spaces* in order to obtain provably larger convex admissible subsets, giving significantly more efficient monitoring schemes. In a nutshell, our proposed *Convex Decomposition (CD)* method works by identifying convex subsets of the *inadmissible region*, and using them to define non-redundant collections of half-spaces that separate these subsets from the admissible region [27]. Our CD methodology can be applied to several important approximate query monitoring tasks (e.g., norms, range aggregates, and joins) giving provably larger SZs and substantially better performance than the original GM.

5 Conclusions and Future Directions

We have given a brief introduction to the distributed data streaming model and the Geometric Method (GM), a generic technique for effectively tracking complex queries over massive distributed streams. We have also discussed recently-proposed extensions to the basic GM framework, such as the combination with AMS stream sketches and local prediction models, as well as recent developments leading to a more general theory of *Safe Zones* for geometric monitoring and interesting connections to convex Euclidean geometry. The GM framework provides a very powerful tool for dealing with continuous query computations over distributed streaming data; see, for instance, [33] for a novel application of the GM to continuous monitoring of skyline queries over fragmented dynamic data.

Continuous distributed streaming is a vibrant, rapidly evolving field of research, and a community of researchers has started forming around theoretical, algorithmic, and systems issues in the area [31] Naturally, there are several promising directions for future research. First, the single-level hierarchy model (depicted in Figure 1(a)) is simplistic and also introduces a single point of failure (i.e., the coordinator). Extending the model to general hierarchies is probably not that difficult (even though effectively distributing the error bounds across the internal hierarchy nodes can be challenging [7]); however, extending the ideas to general, scalable distributed architectures (e.g., P2P networks) raises several theoretical and practical challenges. Second, while most of the proposed algorithmic tools have been prototyped and tested with real-life data streams, there is still a need for real system implementations that also address some of the key systems questions that arise (e.g., what functions and query language to support, how to interface to real users and applications, and so on). We have already started implementing some of the geometric monitoring ideas using Twitter's Storm/λ-architecture, and exploiting these ideas for large-scale, distributed Complex Event Processing (CEP) in the context of the FERARI project (`www.ferari-project.eu`). Finally, from a more foundational perspective, there is a need for developing new models and theories for studying the complexity of such *continuous distributed computations*. These could build on the models of *communication complexity* [26] that study the complexity of distributed *one-shot* computations, perhaps combined with relevant ideas from information theory (e.g., distributed source coding). Some initial results in this direction have recently appeared for the case of simple norms and linear aggregates, e.g., [12, 23].

Acknowledgements. This work was partially supported by the European Commission under ICT-FP7-FERARI (Flexible Event Processing for Big Data Architectures), `www.ferari-project.eu`.

References

[1] N. Alon, P. B. Gibbons, Y. Matias, and M. Szegedy. "Tracking Join and Self-Join Sizes in Limited Storage". In *Proc. of the 18th ACM Symposium on Principles of Database Systems*, May 1999.

[2] N. Alon, Y. Matias, and M. Szegedy. "The Space Complexity of Approximating the Frequency Moments". In *Proc. of the 28th Annual ACM Symposium on the Theory of Computing*, May 1996.

[3] B. Babcock and C. Olston. "Distributed Top-K Monitoring". In *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, June 2003.

[4] S. Burdakis and A. Deligiannakis. "Detecting Outliers in Sensor Networks Using the Geometric Approach". In *Proc. of the 28th Intl. Conference on Data Engineering*, Apr. 2012.

[5] M. Charikar, K. Chen, and M. Farach-Colton. "Finding Frequent Items in Data Streams". In *Proc. of the Intl. Colloquium on Automata, Languages, and Programming*, July 2002.

[6] G. Cormode and M. Garofalakis. "Streaming in a Connected World: Querying and Tracking Distributed Data Streams". Tutorial in *2007 ACM SIGMOD Intl. Conf. on Management of Data*, June 2007.

[7] G. Cormode and M. Garofalakis. "Approximate Continuous Querying over Distributed Streams". *ACM Transactions on Database Systems*, 33(2), June 2008.

[8] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". *Foundations and Trends in Databases*, 4(1-3), 2012.

[9] G. Cormode, M. Garofalakis, S. Muthukrishnan, and R. Rastogi. "Holistic Aggregates in a Networked World: Distributed Tracking of Approximate Quantiles". In *Proc. of the 2005 ACM SIGMOD Intl. Conference on Management of Data*, June 2005.

[10] G. Cormode, M. Garofalakis, and D. Sacharidis. "Fast Approximate Wavelet Tracking on Streams". In *Proc. of the 10th Intl. Conference on Extending Database Technology (EDBT'2006)*, Mar. 2006.

[11] G. Cormode and S. Muthukrishnan. "What's Hot and What's Not: Tracking Most Frequent Items Dynamically". In *Proc. of the 22nd ACM Symposium on Principles of Database Systems*, June 2003.

[12] G. Cormode, S. Muthukrishnan, and K. Yi. "Algorithms for distributed functional monitoring". *ACM Transactions on Algorithms*, 7(2), 2011.

[13] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk. "Gigascope: A Stream Database for Network Applications". In *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, June 2003.

[14] A. Das, S. Ganguly, M. Garofalakis, and R. Rastogi. "Distributed Set-Expression Cardinality Estimation". In *Proc. of the 30th Intl. Conference on Very Large Data Bases*, Sept. 2004.

[15] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. "Processing Complex Aggregate Queries over Data Streams". In *Proc. of the 2002 ACM SIGMOD Intl. Conference on Management of Data*, June 2002.

[16] S. Ganguly, M. Garofalakis, and R. Rastogi. "Processing Set Expressions over Continuous Update Streams". In *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, June 2003.

[17] M. Garofalakis, D. Keren, and V. Samoladas. "Sketch-based Geometric Monitoring of Distributed Stream Queries". In *Proc. of the 39th Intl. Conference on Very Large Data Bases*, Aug. 2013.

[18] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster. "Prediction-based Geometric Monitoring of Distributed Data Streams". In *Proc. of the 2012 ACM SIGMOD Intl. Conference on Management of Data*, May 2012.

[19] N. Giatrakos, A. Deligiannakis, M. Garofalakis, I. Sharfman, and A. Schuster. "Distributed Geometric Query Monitoring using Prediction Models". *ACM Transactions on Database Systems*, 39(2), May 2014.

[20] P. B. Gibbons. "Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports". In *Proc. of the 27th Intl. Conference on Very Large Data Bases*, Sept. 2001.

[21] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. "How to Summarize the Universe: Dynamic Maintenance of Quantiles". In *Proc. of the 28th Intl. Conference on Very Large Data Bases*, Aug. 2002.

[22] M. B. Greenwald and S. Khanna. "Space-Efficient Online Computation of Quantile Summaries". In *Proc. of the 2001 ACM SIGMOD Intl. Conference on Management of Data*, May 2001.

[23] Z. Huang, K. Yi, and Q. Zhang. "Randomized algorithms for tracking distributed count, frequencies, and ranks". In *Proc. of the 31st ACM Symposium on Principles of Database Systems*, May 2012.

[24] R. Keralapura, G. Cormode, and J. Ramamirtham. "Communication-efficient distributed monitoring of thresholded counts". In *Proc. of the 2006 ACM SIGMOD Intl. Conference on Management of Data*, June 2006.

[25] D. Keren, I. Sharfman, A. Schuster, and A. Livne. "Shape-Sensitive Geometric Monitoring". *IEEE Transactions on Knowledge and Data Engineering*, 24(8), Aug. 2012.

[26] E. Kushilevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[27] A. Lazerson, I. Sharfman, D. Keren, A. Schuster, M. Garofalakis, and V. Samoladas. "Monitoring Distributed Streams using Convex Decompositions". In *Proc. of the 41st Intl. Conference on Very Large Data Bases*, Aug. 2015.

[28] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. "The Design of an Acquisitional Query Processor for Sensor Networks". In *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, June 2003.

[29] G. S. Manku and R. Motwani. "Approximate Frequency Counts over Data Streams". In *Proc. of the 28th Intl. Conference on Very Large Data Bases*, Aug. 2002.

[30] S. Muthukrishnan. "Data Streams: Algorithms and Applications". *Foundations and Trends in Theoretical Computer Science*, 1(2), 2005.

[31] NII Shonan Workshop on Large-Scale Distributed Computation, Shonan Village, Japan, January 2012. `http://www.nii.ac.jp/shonan/seminar011/`.

[32] C. Olston, J. Jiang, and J. Widom. "Adaptive Filters for Continuous Queries over Distributed Data Streams". In *Proc. of the 2003 ACM SIGMOD Intl. Conference on Management of Data*, June 2003.

[33] O. Papapetrou and M. Garofalakis. "Continuous Fragmented Skylines over Distributed Streams". In *Proc. of the 30th Intl. Conference on Data Engineering*, Apr. 2014.

[34] I. Sharfman, A. Schuster, and D. Keren. "A geometric approach to monitoring threshold functions over distributed data streams". In *Proc. of the 2006 ACM SIGMOD Intl. Conference on Management of Data*, June 2006.

Data Engineering

It's FREE to join!

TCDE

tab.computer.org/tcde/

The Technical Committee on Data Engineering (TCDE) of the IEEE Computer Society is concerned with the role of data in the design, development, management and utilization of information systems.

- Data Management Systems and Modern Hardware/Software Platforms
- Data Models, Data Integration, Semantics and Data Quality
- Spatial, Temporal, Graph, Scientific, Statistical and Multimedia Databases
- Data Mining, Data Warehousing, and OLAP
- Big Data, Streams and Clouds
- Information Management, Distribution, Mobility, and the WWW
- Data Security, Privacy and Trust
- Performance, Experiments, and Analysis of Data Systems

The TCDE sponsors the International Conference on Data Engineering (ICDE). It publishes a quarterly newsletter, the Data Engineering Bulletin. If you are a member of the IEEE Computer Society, you may join the TCDE and receive copies of the Data Engineering Bulletin without cost. There are approximately 1000 members of the TCDE.

Join TCDE via Online or Fax

ONLINE: Follow the instructions on this page:

www.computer.org/portal/web/tandc/joinatc

FAX: Complete your details and fax this form to **+61-7-3365 3248**

Name _____

IEEE Member # _____

Mailing Address _____

_____

Country _____

Email _____

Phone _____

TCDE Mailing List

TCDE will occasionally email announcements, and other opportunities available for members. This mailing list will be used only for this purpose.

Membership Questions?

Xiaofang Zhou
School of Information Technology and Electrical Engineering
The University of Queensland
Brisbane, QLD 4072, Australia
zxf@uq.edu.au

TCDE Chair

Kyu-Young Whang
KAIST
371-1 Koo-Sung Dong, Yoo-Sung Ku
Daejeon 305-701, Korea
kywhang@cs.kaist.ac.kr

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398