Title
Queries with Bounded Errors & Bounded Response Times on Very Large Data

Author
Agarwal, Sameer

Publication Date
2014

Peer reviewed|Thesis/dissertation

Queries with Bounded Errors & Bounded Response Times on Very Large Data

by

Sameer Agarwal

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ion Stoica, Chair
Professor Joshua S. Bloom
Professor Joseph M. Hellerstein
Professor Scott J. Shenker

Fall 2014

Queries with Bounded Errors & Bounded Response Times on Very Large Data

by

Sameer Agarwal

Abstract

Queries with Bounded Errors & Bounded Response Times on Very Large Data

by

Sameer Agarwal

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Modern data analytics applications typically process massive amounts of data on clusters of tens, hundreds, or thousands of machines to support near-real-time decisions. The quantity of data and limitations of disk and memory bandwidth often make it infeasible to deliver answers at human-interactive speeds. However, it has been widely observed that many applications can tolerate some degree of inaccuracy. This is especially true for *exploratory queries* on data, where users are satisfied with "close-enough" answers if they can be provided quickly to the end user. A popular technique for speeding up queries at the cost of accuracy is to execute each query on a sample of data, rather than the whole dataset. In this thesis, we present BlinkDB, a massively parallel, approximate query engine for running interactive SQL queries on large volumes of data. BlinkDB allows users to trade-off query accuracy for response time, enabling interactive queries over massive data by running queries on data samples and presenting results annotated with meaningful error bars. To achieve this, BlinkDB uses three key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements, and (3) an error estimation and diagnostics module that produces approximate answers and reliable error bars. We evaluate BlinkDB extensively against well-known database benchmarks and a number of real-world analytic workloads showing that it is possible to implement an end-to-end query approximation pipeline that produces approximate answers with reliable error bars at interactive speeds.

To my family,
without whom nothing
would be much worth doing

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I would like to express my sincere gratitude towards my advisor, Ion Stoica, for his unconditional support and guidance throughout my graduate studies. Ion helped me realize the importance of choosing and formulating the right problems and taught me the invaluable skill of separating insights from ideas. Even after all these years, Ion continues to amaze me with his infectious enthusiasm and his relentless drive for perfection.

The genesis of this thesis can be traced all the way back to early 2011 when Sam Madden was on a sabbatical at the AMPLab in UC Berkeley. Since then, Sam has been an amazing collaborator and an excellent sounding board for many ideas that have now become chapters in this thesis. I am also extremely grateful to Srikanth Kandula for being a close collaborator and a great mentor. Srikanth has been a constant source of inspiration over the years and has always motivated me to aim higher.

I am indebted to my dissertation committee members — Professors Josh Bloom, Joe Hellerstein and Scott Shenker for their detailed feedback and thoughtful comments during the course of this thesis. In particular, I am really grateful to Joe for sharing my excitement in this area of research and for his extremely useful suggestions on many technical aspects of this thesis. I am thankful to Josh for helping me look at this work from a larger perspective and to Scott for his crucial advice on several occasions during the course of my graduate studies. I am also grateful to Scott Shenker and Sylvia Ratnasamy for giving me the opportunity to assist them in teaching various courses at Berkeley. Seeing their enthusiasm and dedication towards their students will continue to inspire me for many years to come.

I would also like to thank professors Ken Birman, Gautam Barua, Diganta Goswami and Purandar Bhaduri for nurturing my interest in Computer Science and for encouraging me to pursue higher studies. I am really grateful to Mahesh Balakrishnan for exposing me to the exciting world of Systems research. If it were not for their firm belief in my capabilities, I would not have been here.

I am also grateful for the PhD fellowships from Qualcomm and Facebook Research during the course of my graduate studies. These generous fellowships not only gave me the freedom to choose and pursue interesting problems but were also accompanied with amazing mentorship. In particular, I would like to thank Michael Weber, Seth Fowler and Dilma Da Silva from Qualcomm Research, and Ravi Murthy, Martin Traverso and Dain Sundstrom from Facebook for many insightful discussions that inspired many aspects of my research.

This thesis would not have been possible without the constant influx of fresh ideas from a number of collaborators, friends and mentors who continue to inspire and guide me. In particular, I would like to express my sincere gratitude towards Ganesh Ananthanarayanan, Nicolas

Bruno, Mosharaf Chowdhury, Anand Iyer, Ariel Kleiner, Sanjay Krishnan, Henry Milner, Barzan Mozafari, Aurojit Panda, Purna Sarkar, Jun Suzuki, Ameet Talwalkar, Shivaram Venkataraman, Jiannan Wang, David Zats, Jingren Zhou, and Professors Michael Jordan, Michael Franklin and Marti Hearst for extremely productive collaborations and advice.

Outside of AMPLab, life at Berkeley would not have been the same if it were not for the amazing set of friends I had. In particular, I would like to thank Mangesh Bangar, Avinash Bhardwaj, Rutooj Deshpande, Ashley D'Mello, Kartik Ganapathi, Sarika Goel, Gautam Gundiah, Gagan Gupta, Ankit Jain, Tim Ketron, Shaama M.S., Manali Nekkanti, Ram Nekkanti, Subha Srinivasan and Sajad Zafranchilar for all the fun times we had.

Last but not least, I am deeply indebted to my family — my parents, my grandparents and my little brother, Saurabh — for their unconditional love, support and understanding over the years. My grandfather, whom I deeply miss, helped me believe that I can achieve more than I ever dreamed I could. This work is a dedication to their dreams and aspirations.

Chapter 1

Introduction

Modern data analytics applications typically involve computing aggregates over a large number of records to *roll up* web clicks, online transactions, content downloads, and hundreds of other features along a variety of different dimensions, including demographics, content type, region, and so on. Traditionally, such queries are executed using sequential scans over a large fraction of a database. However, increasingly, new applications demand near real-time response rates. While the examples of these applications are wide and varied, the delay in response times can be attributed to two main causes. Firstly, when processing data on the order of petabytes, algorithmic complexities start to manifest at time-scales that matter to humans, and combined with other systems level issues, such large processing times can significantly impede the ability for people to gain any meaningful feedback from the workflow. Secondly, I/O rates place a bound on the processing rate. Scanning even a few terabytes of data stored in a modern distributed file system can take in the order of minutes, despite the data being "striped" across hundreds of disks or cached in memory. While the amount of data makes arbitrary interactive queries prohibitive, it has been widely observed that *many analytic queries can tolerate some levels of inaccuracy*. This is especially true for the kinds of exploratory queries one might run on data initially where a user might be able to derive the same benefit from a *close enough* answer instead of an exact one.

For instance, imagine a situation where one is attempting to debug a large-scale distributed application. Such debugging is often done by analyzing logs, and carrying out root-cause analysis. These logs are often very large, as programmers and service providers attempt to balance the amount of information stored in logs, versus the amount of data that has to be read to carry out any analysis. Furthermore, the process of analysis is often fairly complex. For example, consider a situation when a subset of users of an online video streaming website such as Netflix [61] experience quality issues — for instance, a high level of buffering or large start-up times. Diagnosing such problems needs to be done quickly to avoid lost revenue, and the causes can be varied. For example, the Content Distribution Network (`CDN`) in a certain geographic region may be overloaded, an Internet Service

Provider (ISP) may be experiencing high level of congestion, a firmware upgrade of a set-top box may not be able to sustain a sufficient bit rate or a new version of the video player may exhibit bad interaction with the browser on a particular mobile platform. Whatever may be the cause, from the database perspective, diagnosing such problems requires rolling up data across tens of dimensions to find the particular attributes (e.g., client OS, browser, firmware, device, geo location, ISP, CDN or content) that best characterize the users experiencing the problem. Diagnosing such problems quickly is essential, and speed is often more important than *exact* results. Other examples in this category may include detecting spam in a social networking service such as Twitter [76], detecting denial of service attacks in an ISP, or detecting intrusions in an organization.

In another example, consider an online business that wants to adapt its policies and decisions in near real time to maximize its ad revenue. This might again involve aggregating across multiple dimensions to understand how an ad performs given a particular group of users, content, site, and time of day. Performing such analysis quickly is essential, especially when there is a change in the environment, e.g., new ads, new content or a new page layout. The ability to re-optimize the ad placement every minute as opposed to every day or week often leads to a material difference in revenue.

To further drive home the point, consider an online service company that aims to optimize its business by improving user retention, or by increasing their user engagement. Often this is done by using A/B testing to experiment with anything from new products to slight changes in the web page layout, format, or colors[1]. The number of combinations and changes that one can test is daunting, even for fairly large companies with clusters of hundreds or thousands of machines at their disposal. Furthermore, such tests need to be conducted carefully as they may negatively impact the user experience. The ability to quickly understand the impact of various tests and identify important trends is critical to rapidly improving the business.

In these and many other analytic applications, queries are unpredictable (because the exact problem, or query is not known in advance) and quick response time is essential as data is changing quickly, and the potential profit (or loss in profit in the case of service outages) is inversely proportional to the response time. Another common characteristic of all of the above applications is that they compute an aggregate view of certain metrics, and are interested in how they change over time. Analysis of this form is often dependent only on observations about anomalously different values, rather than an exact quantity. We focus specifically on these applications.

[1] A much publicized example is Google's testing with 41 shades of blue for their home page [67].

1.1 Achieving Interactive Response Times

The conventional way of answering such queries requires scanning the entirety of several terabytes or petabytes of data which can be quite inefficient. For example, computing a simple average over 10 terabytes of data stored on 100 machines can take several minutes if the data is striped on disks, and tens or hundreds of seconds even if the entire data is cached in memory. This is unacceptable for rapid problem diagnosis, and frustrating even for exploratory analysis. There are two possible ways of getting around this problem.

Adding Resources and Optimizing Query Plans

One way to decrease the overall query response time is to add more resources (i.e., *memory* or *CPU*) and/or to optimize (or sometimes constrain) the query execution plan. Recent work on data-parallel cluster computing frameworks has mainly focused on solving issues that arise during the execution of jobs, by leveraging more memory [84, 81] and/or parallelism [12, 57], effectively sharing the cluster [49, 72], tackling outliers [11], guaranteeing data locality [82], optimizing network communication [32] or incorporating new functionality such as the support for iterative and recursive control flow [60].

In the database community, there has been extensive work on optimizing query plans based on the underlying data. The AutoAdmin [31] project examined adapting physical database design (e.g., choosing which indices to build and which views to materialize, based on the data and queries). Kabra and DeWitt [51] were ones of the earliest to propose a scheme that collects statistics, re-runs the query optimizer concurrently with the query, and migrates from the current query plan to an improved one, if doing so is predicted to improve performance. Eddies [16] adapts query executions at a much finer, per-tuple, granularity. Starfish [48] examines Hadoop jobs, one map followed by one reduce, and tunes low-level configuration variables in Hadoop [15] by constructing a *what-if* engine based on a classifier trained on experiments over a wide range of parameter choices.

While a number of these systems deliver low-latency response times when each node has to process a relatively small amount of data (e.g., when the data can fit in the aggregate memory of the cluster), they become slower as the data grows unless new resources are constantly being added in proportion. Additionally, a significant portion of query execution time in these systems involves shuffling or re-partitioning massive amounts of data over the network, which is often a bottleneck for queries.

Trading-*off* Accuracy

Another way to achieve interactivity is to provide quick approximate answers. Approximate Query Processing (AQP) for decision support in relational databases has been the subject of extensive research, and has effectively leveraged samples, or other non-sampling based

approaches. One can therefore envision a system where one samples from the underlying data, and uses approximate values to answer queries quickly, thus giving the perception of nearly interactive queries while allowing analysts to process large quantities of data that is often essential for a number of use-cases. Our conversations with a large number of companies, both big and small, have led us to believe that users often employ ad-hoc heuristics to obtain faster response times, such as selecting small slices of data (e.g., an hour) or arbitrarily sampling the data [25, 69]. These efforts suggest that, at least in many analytic applications, users are willing to forgo accuracy for achieving better response times. Despite these potential benefits, no approximate query processing system is in wide use today. We believe this can be explained by two problems that no existing system has overcome.

First, sampling introduces error, and users always want to know how much error has been introduced– i.e., "what do the error bars look like?" Complementarily, the system needs to know the errors to determine an appropriate sample size. Unfortunately, we discovered that existing techniques for computing error bars are too often inaccurate when applied to real analytical queries. While they are based on solid results from the statistics literature, these results require assumptions that are violated in practice, or else are too conservative to be useful. If the error bars cannot be trusted, the system cannot be used.

Second, even given accurate error bars, users are reluctant to accept some possibility of error. This is perhaps reasonable; an approximate answer with accompanying error bars is harder to use than an exact one. One way to overcome this reluctance is to provide a compelling reason to use an AQP system – a new use case, not just a speedup. AQP systems have the potential to answer *complex* analytic queries on large datasets at *interactive speeds* suitable for rapid refinement by human analysts. By "complex" queries we mean queries that compute more than simple `SUM`s or `COUNT`s; for example, today's analysts often use user-defined functions (UDFs) that execute arbitrary Java code over an iterator. Existing systems have supported only simple queries, have not been demonstrated to work at scale, or have not delivered interactive speeds.

1.2 Making Approximate Queries Practical

Approximate Query Processing (AQP) has a long history in databases. Nearly three decades ago, Olken and Rotem [62] introduced random sampling in relational databases as a means to return approximate answers and reduce query response times. A large body of work has subsequently proposed different sampling techniques [1, 5, 28, 35, 39, 47, 53, 66, 68]. All of this work shares the same motivation: executing queries on precomputed samples (or more generally, any subset of data) can dramatically improve the latency and resource costs of queries. Indeed, sampling can produce a *greater-than-linear* speedup in query response time if the sample can fit into memory but the whole dataset cannot, enabling a system to return answers in only a few seconds. Research on human-computer interactions, famously the work

of Miller [58], has shown that such quick response times can make a qualitative difference in user interactions.

With no approximate query processing framework in wide use today, we created one from scratch and called it BlinkDB [4, 5]. BlinkDB is a distributed sampling-based approximate query processing framework that strives to make approximate queries practical by achieving a better balance between efficiency and generality for analytic workloads. We extensively leverage and build on the past three decades of database research on approximate query processing and over a century of statistical research on error estimation [45, 70] to make approximate query processing frameworks more practical, more usable and most importantly, trustworthy.

BlinkDB allows users to pose SQL-based aggregation queries (those with `COUNT, AVG, SUM, VARIANCE, QUANTILES`), or any `UDF` (i.e., a user defined function) over stored data, along with response time or error bound constraints. As a result, queries over multiple terabytes of data can be answered in seconds, accompanied by meaningful error bounds relative to the answer that would be obtained if the query ran on the full data. In contrast to many existing approximate query solutions (e.g., [28]), BlinkDB supports more general queries as it makes no assumptions about the attribute values in the `WHERE`, `GROUP BY`, and `HAVING` clauses, or the distribution of the values used by aggregation functions. Instead, BlinkDB only assumes that the sets of columns used by queries in `WHERE`, `GROUP BY`, and `HAVING` clauses are stable over time. We call these sets of columns *"query column sets"* or QCSs in this thesis.

Query Interface

As an example, let us consider querying a table `Sessions`, with five columns, `SessionID`, `Genre`, `OS`, `City`, and `URL`, to determine the number of sessions in which users viewed content in the "`western`" genre, grouped by `OS`. The query:

```
SELECT COUNT(*)
FROM Sessions
WHERE Genre = 'western'
GROUP BY OS
ERROR WITHIN 10% AT CONFIDENCE 95%
```

in BlinkDB will return the count for each `GROUP BY` key, with each count having relative error of at most $\pm 10\%$ at a 95% confidence level. Alternatively, a query of the form:

```
SELECT COUNT(*)
FROM Sessions
WHERE Genre = 'western'
GROUP BY OS
WITHIN 5 SECONDS
```

in BlinkDB will return the most accurate results for each `GROUP BY` key in 5 seconds, along with a 95% confidence interval for the relative error of each result.

We are motivated by a desire to provide users with the ability to trade away accuracy for savings in time and energy, and show that many classes of problems can take advantage of this trade-off. We believe that for many classes of problems, users would rather have approximate, but immediate answers, rather than getting precise, stale answers. BlinkDB[2] has been open-sourced and has been deployed in production clusters at Facebook Inc. and a number of other companies.

1.3 Dissertation Overview

The rest of this thesis is organized into 7 chapters. Chapter 2 focuses on *sampling* and discusses both the initial sample selection, and sample maintenance. BlinkDB uses both uniform random samples, and a number of biased samples stratified across a number of dimensions. For uniform sampling we use information about how sample size affects variance of queries to determine what sample sizes to maintain. For instance we observe that sample mean varies as the square root of sample size (i.e., $\sigma_m \propto 1/\sqrt{n}$), and hence the effects of increasing sample size decline dramatically as sample sizes themselves increase (in particular, the function is asymptotic), and hence we choose to have a larger number of small samples as opposed to large samples. For biased sampling, we use a MILP optimization function that leverages historic information to determine what column or columns we should bias our samples using.

Chapter 3 focuses on *runtime sample selection* and is responsible for determining the cost of running any particular query for picking the best sample a query should run against. While for built-in operators we can use known cost functions, we find that the use of user defined functions is widespread in these frameworks, and hence we consider other methods of estimating query costs.

Approximate answers are most useful when accompanied by accuracy guarantees. Therefore, a key aspect of almost all AQP systems is their ability to *estimate* the *error* of their returned results. Most commonly, error estimates come in the form of confidence intervals

[2]http://blinkdb.org

(a.k.a. "error bars") that provide bounds on the error caused by sampling. Such error estimates allow the AQP system to check whether its sampling method produces results of reasonable accuracy. They can also be reported directly to users, who can factor the uncertainty of the query results into their analyses and decisions. Further, error estimates help the system control error: by varying the sample size while estimating the magnitude of the resulting error bars, the system can make a smooth and controlled trade-off between accuracy and query time. For these reasons, many methods have been proposed for producing reliable error bars—the earliest being closed-form estimates based on either the central limit theorem (CLT) [70] or on large deviation inequalities such as Hoeffding bounds [45]. Unfortunately, deriving closed forms is often a manual, analytical process. As a result, closed-form-based S-AQP systems [1, 5, 28, 35, 39, 47, 68] are restricted to very simple SQL queries (often with only a single layer of basic aggregates like `AVG, SUM, COUNT, VARIANCE` and `STDEV` with projections, filters, and a `GROUP BY`). This has motivated the use of resampling methods like the bootstrap [53, 66], which require no such detailed analysis and can be applied to arbitrarily complex SQL queries. In exchange, the bootstrap adds some additional overhead. Chapter 4 focuses on these statistical aspects of error estimation.

Beyond these computational considerations, it is critical that the produced error bars be *reliable*, i.e., that they not under- or overestimate the actual error. Underestimating the error misleads users with a false confidence in the approximate result, which can propagate to their subsequent decisions. Overestimating the error is also undesirable. An overestimate of error forces the S-AQP system to use an unnecessarily large sample, even though it could achieve a given level of accuracy using a much smaller sample, and hence much less computation. Approximation is most attractive when it is achieved with considerably less effort than needed for fully accurate results, so inflating sample sizes is problematic. The past decade has seen several investigations of such *diagnostic* methods in the statistics literature [26, 52].

However, unfortunately, no existing error estimation technique is ideal. Large deviation inequalities (e.g., Hoeffding bounds used in [1, 47]) can provide very loose bounds in practice [47], leading to overestimation of error and thus an unnecessary increase in computation. For instance, in Fig. 1.1 we show the sample sizes needed to achieve different levels of relative error (averaged over 100 Hive queries from production clusters at Conviva Inc. on tens of terabytes of data[3] with vertical bars on each point denoting 0.01 and 0.99 quantiles). If the AQP system were to believe the error estimates of these different techniques, a system relying on Hoeffding bounds must use samples that are 1–2 orders of magnitude larger than what is otherwise needed, diminishing the performance benefits of approximation. On the other hand, CLT-based and bootstrap-based error estimation methods—while being significantly more practical than large deviation inequalities—do not always yield accurate error estimates either. These techniques are guaranteed to work well only under regularity assumptions (in particular, assumptions on the *smoothness* of query aggregates) that are sometimes unrealis-

[3] A more detailed description of our datasets and experiments is presented in Chapter 6.

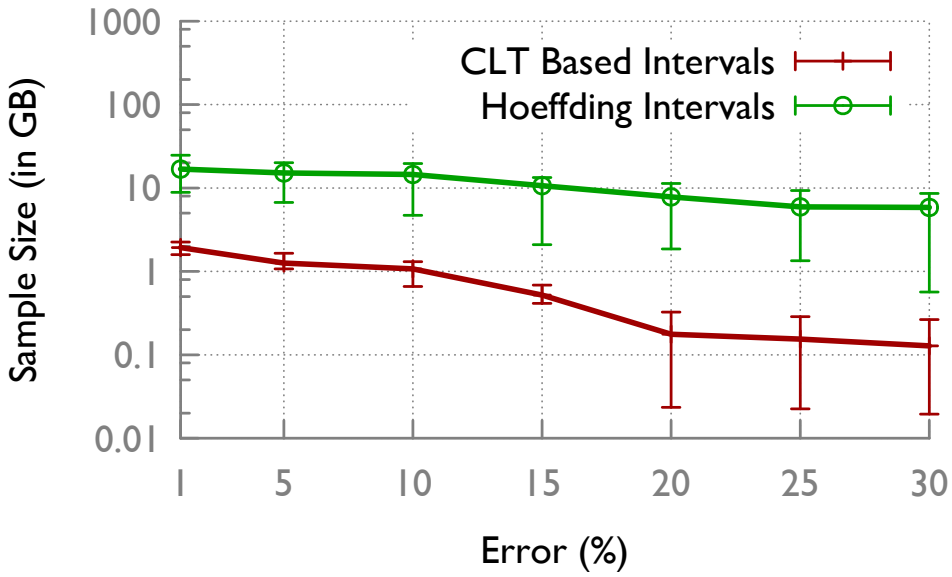


Figure 1.1: Sample sizes suggested by different error estimation techniques for achieving different levels of relative error.

tic and can produce both underestimates and overestimates in practice. Chapter 5 suggests a diagnostic algorithm to validate multiple procedures for generating error bars at runtime and present a series of optimization techniques across all stages of the query processing pipeline that reduce the running time of the diagnostic algorithm from hundreds of seconds to only a couple of seconds, making it a practical tool in a distributed AQP system.

Chapter 6 then discusses several optimizations that make the diagnostics and the procedures that generate error bars practical, ensuring that these procedures do not affect the interactivity of the overall query. With these optimizations in place, and by leveraging recent systems for low-latency exact query processing, we demonstrate a viable end-to-end system for approximate query processing using sampling. We show that this system can deliver interactive-speed results for a wide variety of analytic queries from real world production clusters. Finally, we conclude in Chapter 7.

Chapter 2

Sampling

Over the past few decades a large number of approximation techniques [38] have been proposed, which allow for fast processing of large amounts of data by trading result accuracy for response time and space. These techniques include sampling [43, 28, 8, 13, 14, 18, 33, 47], sketches [37], histograms [19, 20, 29, 22, 23, 24] and wavelets [27]. To illustrate the utility of such techniques, consider the following simple query that computes the average `SessionTime` over all users originating in `New York`:

```
SELECT AVG(SessionTime)
FROM Sessions
WHERE City = 'New York'
```

Suppose the `Sessions` table contains 100 million tuples for `New York`, and cannot fit in memory. In that case, the above query may take a long time to execute, since disk reads are expensive, and such a query would need multiple disk accesses to stream through all the tuples. Suppose we instead executed the same query on a precomputed sample containing only $10,000$ `New York` tuples, such that the entire sample fits in memory. This would be orders of magnitude faster, while still providing an approximate result within a few percent of the actual value, an accuracy good enough for many practical purposes. Using sampling theory we could even provide confidence bounds on the accuracy of the answer [55].

Previously described approximation techniques make different trade-offs between efficiency and the generality of the queries they support. At one end of the spectrum, existing solutions based on pre-computed samples and/or sketches exhibit low space and time complexity, but typically make strong assumptions about the query workload (e.g., they assume they know the set of tuples accessed by future queries and aggregation functions used in queries). As an example, if we know all future queries are on large cities, we could simply pre-compute random samples that omit data about smaller cities.

At the other end of the spectrum, computing samples at query runtime [47] makes little or no assumptions about the query workload, at the expense of highly variable performance. Using these methods, the above query will likely finish much faster for sessions in `New York` (i.e., the user might be satisfied with the result accuracy, once the query sees the first $10,000$ sessions from `New York`) than for sessions in `Galena, IL`, a town with fewer than $4,000$ people. In fact, for such a small town, these methods may need to read the entire table to compute a result with satisfactory error bounds.

In this chapter, we argue that none of the previous solutions alone are a good fit for today's big data analytics workloads. Computing samples at runtime provides relatively poor performance for queries on rare tuples, while current sampling and sketch based techniques make strong assumptions about the predictability of workloads or substantially limit the types of queries they can execute. In contrast to most existing approximate query solutions (e.g., [28]), in BlinkDB, we support more general queries as it makes no assumptions about the attribute values in the `WHERE`, `GROUP BY`, and `HAVING` clauses, or the distribution of the values used by aggregation functions. Instead, BlinkDB only assumes that the sets of columns used by queries in `WHERE`, `GROUP BY`, and `HAVING` clauses are stable over time. Recall that we call these sets of columns *"query column sets"* or QCSs.

2.1 Background

First and foremost, any offline-sampling based query processor, including BlinkDB, must decide what types of samples to create. The sample creation process must make some assumptions about the nature of the future query workload. One common assumption is that future queries will be similar to historical queries. While this assumption is broadly justified, it is necessary to be precise about the meaning of "similarity" when building a workload model. A model that assumes the wrong kind of similarity will lead to a system that "over-fits" to past queries and produces samples that are ineffective at handling future workloads. This choice of model of past workloads is one of the key differences between sampling in BlinkDB and a lot of prior work. In the rest of this section, we present a taxonomy of workload models, discuss our approach, and show that it is reasonable using experimental evidence from a production system.

Workload Taxonomy

Offline sample creation, caching, and virtually any other type of database optimization assumes a target workload that can be used to predict future queries. Such a model can either be trained on past data, or based on information provided by users. This can range from an ad-hoc model, which makes no assumptions about future queries, to a model which assumes that all future queries are known *a priori*. As shown in Fig. 2.1, we classify possible approaches into one of four categories:

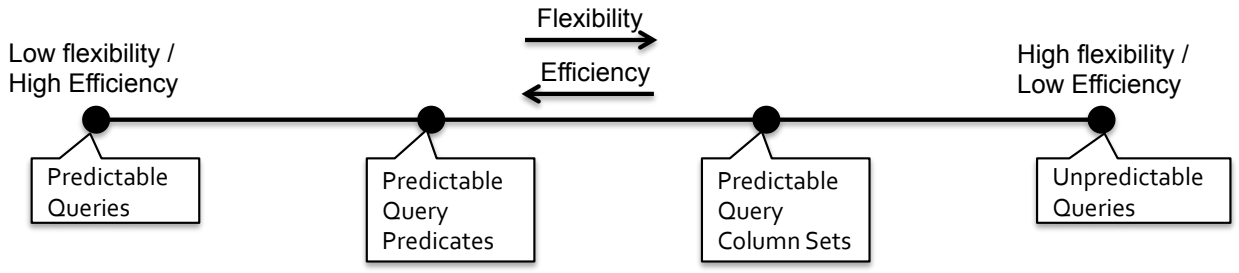


Figure 2.1: Taxonomy of Workload Models.

1. **Predictable Queries:** At the most restrictive end of the spectrum, one can assume that all future queries are known in advance, and use data structures specially designed for these queries. Traditional databases use such a model for lossless synopsis [37] which can provide extremely fast responses for certain queries, but cannot be used for any other queries. Prior work in approximate databases has also proposed using lossy sketches (including wavelets and histograms) [43]. Similarly, there has been a great deal of work on wavelets, histograms, sketches, and lossless summaries (e.g., materialized views and data cubes). In general, these techniques are tightly tied to specific classes of queries. For instance, Vitter and Wang [79] use Haar wavelets to encode a data cube without reading the least significant bits of SUM/COUNT aggregates in a *flat* query[1], but it is not clear how to use the same encoding to answer joins, subqueries, or other complex expressions. Thus, these techniques are most applicable[2] when future queries are known in advance (modulo constants or other minor details).

2. **Predictable Query Predicates:** A slightly more flexible model is one that assumes that the frequencies of group and filter predicates — both the columns and the values in `WHERE, GROUP BY`, and `HAVING` clauses — do not change over time. For example, if 5% of past queries include only the filter `WHERE City = 'New York'` and no other group or filter predicates, then this model predicts that 5% of future queries will also include only this filter. Under this model, it is possible to predict future filter predicates by observing a prior workload. This model is employed by materialized views in traditional databases. Approximate databases, such as STRAT [28] and SciBORQ [73], have similarly relied on prior queries to determine the tuples that are likely to be used in future queries, and to create samples containing them. *STRAT*, for instance, builds a single stratified sample by minimizing the expected relative error of all previous queries, due to which it has to make stronger assumptions about the future queries. Specifically, STRAT assumes that fundamental regions (FRs) of future queries are identical to the FRs of past queries, where *FR* of a query is the exact set of tuples accessed by that

[1] A SQL statement without any nested sub-queries.

[2] Also, note that materialized views can still be too large for real-time processing.

query. *SciBORQ* [73] on the other hand, is a data-analytics framework designed for scientific workloads, which uses special structures, called *impressions*. Impressions are biased samples where tuples are picked based on past query results. SciBORQ targets exploratory scientific analysis. Unfortunately, in many domains including those discussed in Chapter 1, these assumption do not hold. Even queries with slightly different constants can have entirely different FRs/impressions, and thus, having seen one of them does not imply that STRAT or SciBORQ can minimize the error for the other.

3. **Predictable QCSs:** Even greater flexibility is provided by assuming a model where the frequency of the sets of columns used for grouping and filtering does not change over time, but the exact values that are of interest in those columns are unpredictable. We term the columns used for grouping and filtering in a query the *query column set*, or QCS, for the query. For example, if 5% of prior queries grouped or filtered on the QCS `{City}`, this model assumes that 5% of future queries will also group or filter on this QCS, though the particular predicate may vary. This model can be used to decide the columns on which building indices would optimize data access. Prior work [30, 71] has shown that a similar model can be used to improve caching performance in OLAP systems. AQUA [3, 2], an approximate query database based on sampling, uses the QCS model.

4. **Unpredictable Queries:** Finally, the most general model assumes that queries are *unpredictable*. Given this assumption, traditional databases can do little more than just rely on query optimizers which operate at the level of a single query. In approximate databases, this workload model does not lend itself to any "intelligent" sampling, leaving one with no choice but to uniformly sample data. This model is used by the original Online Aggregation (OLA) project [47], which relied on streaming data in random order. Online Aggregation and many of its successors [35, 65] proposed the idea of providing approximate answers which are constantly refined during query execution. It provides users with an interface to stop execution once they are satisfied with the current accuracy. As commonly implemented, the main disadvantage of OLA systems is that they stream data in a random order, which imposes a significant overhead in terms of I/O. Naturally, these approaches cannot exploit the workload characteristics in optimizing the query execution.

While the unpredictable query model is the most flexible one, it provides little opportunity for an approximate query processing system to efficiently sample the data. Furthermore, prior work [35, 65] has argued that OLA performance's on large clusters (the environment on which BlinkDB is intended to run) falls short. In particular, accessing individual rows randomly imposes significant scheduling and communication overheads, while accessing data

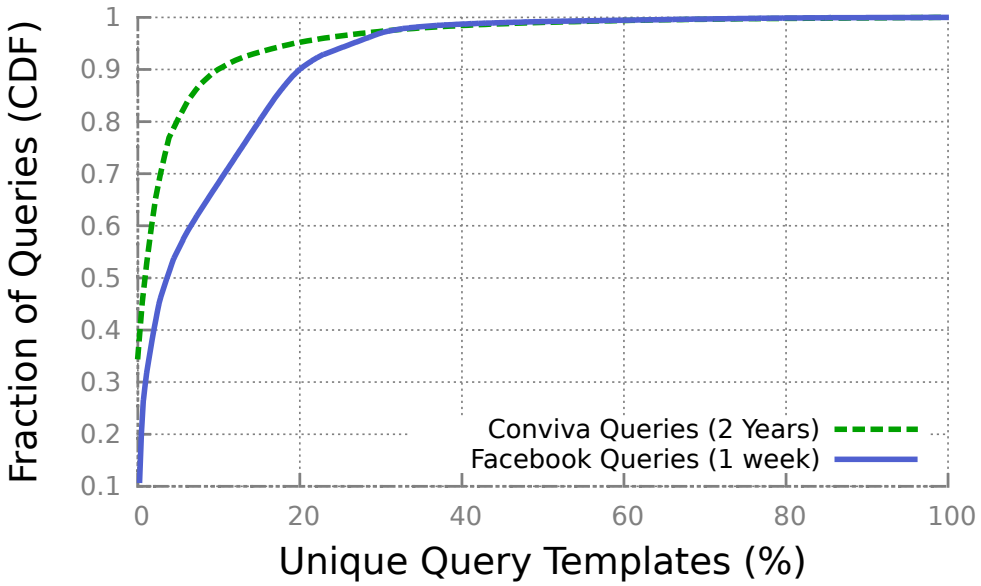


Figure 2.2: Distribution of QCSs across all queries in the Conviva and Facebook traces.

at the HDFS block[3] level may skew the results.

As a result, we use the model of predictable QCSs. As we will show, this model provides enough information to enable efficient pre-computation of samples, and it leads to samples that generalize well to future workloads in our experiments. Intuitively, such a model also seems to fit in with the types of exploratory queries that are commonly executed on large scale analytical clusters. As an example, consider the operator of a video site who wishes to understand what types of videos are popular in a given region. Such a study may require looking at data from thousands of videos and hundreds of geographic regions. While this study could result in a very large number of distinct queries, most will use only two columns, video title and viewer location, for grouping and filtering. Next, we present empirical evidence based on real world query traces from Facebook Inc. and Conviva Inc. to support our claims.

Query Patterns in a Production Cluster

To empirically test the validity of the *predictable* QCS model we analyze a trace of 18,096 queries from 30 days of queries from Conviva and a trace of 69,438 queries constituting a random, but representative, fraction of a 7-day workload from Facebook to determine the frequency of QCSs.

[3]Typically, these blocks are $64 - 1024$ MB in size.

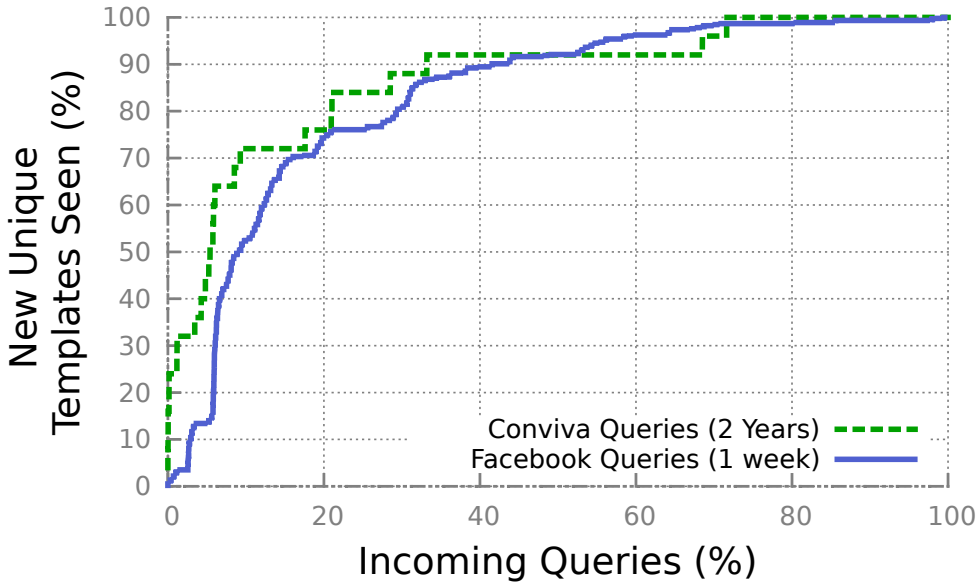


Figure 2.3: Stability of QCSs across all queries in the Conviva and Facebook traces.

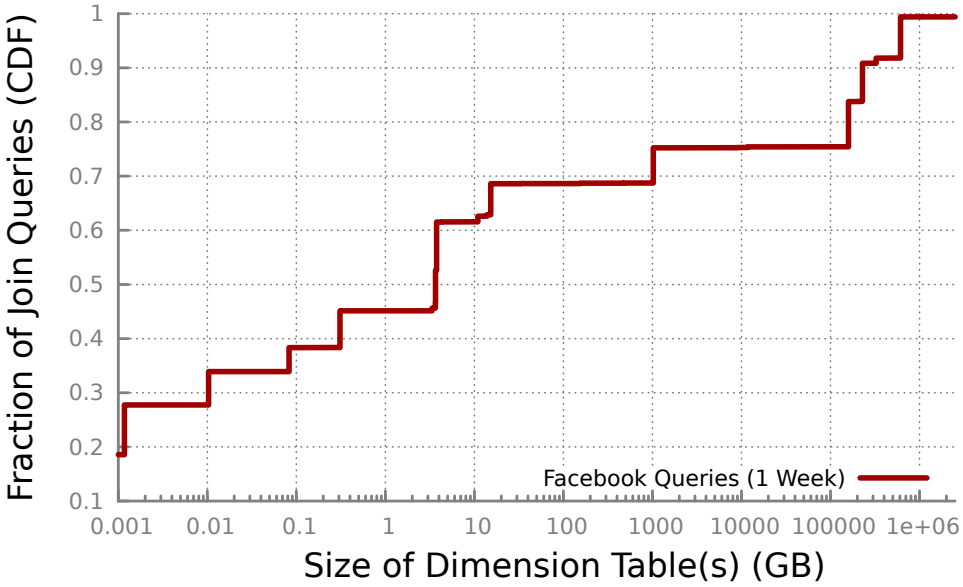


Figure 2.4: CDF of join queries with respect to the size of dimension tables.

Fig. 2.2 shows the distribution of QCSs across all queries for both workloads. Surprisingly, over 90% of queries are covered by 10% and 20% of unique QCSs in the traces from Conviva and Facebook respectively. Only 182 unique QCSs cover all queries in the Conviva trace and 455 unique QCSs span all the queries in the Facebook trace. Furthermore, if we remove

the QCSs that appear in less than 10 queries, we end up with only 108 and 211 QCSs covering 17,437 queries and 68,785 queries from Conviva and Facebook workloads, respectively. This suggests that, for real-world production workloads, QCSs represent an excellent model of future queries.

Fig. 2.3 shows the number of unique QCSs versus the queries arriving in the system. We define unique QCSs as QCSs that appear in more than 10 queries. For the Conviva trace, after only 6% of queries we already see close to 60% of all QCSs, and after 30% of queries have arrived, we see almost all QCSs — 100 out of 108. Similarly, for the Facebook trace, after 12% of queries, we see close to 60% of all QCSs, and after only 40% queries, we see almost all QCSs — 190 out of 211. This shows that QCSs are relatively stable over time, which suggests that the past history is a good predictor for the future workload.

2.2 Leveraging Existing Work

Now that we have presented empirical evidence in favor of a predictable QCSs based approach, one of the most straightforward ways to create a set of optimal samples is by leveraging existing work in the database community. AQUA, as we discussed, creates a single stratified sample for a given table based on the union of the set(s) of columns that occur in the `GROUP BY` or `HAVING` clauses of all the queries on that table. The number of tuples in each *stratum* are then decided according to a weighting function that considers the sizes of groups of all subsets of the grouping attributes. This implies that for g grouping attributes, AQUA considers all 2^g combinations. Unfortunately, this can be prohibitive for large values of g. *Olston et al.* [63] use sampling for interactive data analysis. Their approach requires building a new sample for each QCS, which unfortunately didn't scale too well for large numbers of unique QCSs (e.g., in most of the real-world production workloads, the total number of unique QCSs exceeds 100). *Babcock et al.* [17], on the other hand, describe a stratified sampling technique where biased samples are built on a single column, in contrast to the multi-dimensional samples in AQUA. In their approach, queries are executed on all biased samples whose biased column is present in the query and the union of results is returned as the final answer. While this had much worse error-latency trade-off as compared to AQUA, it only used a fraction of storage space.

On one hand, having multi-dimensional stratified samples produce the best error-latency tradeoff which comes at the cost of huge storage space. On the other hand, limiting ourselves to only single-dimensional stratified samples, while space-efficient, is bad for performance. In order to achieve both, we formulate the problem of sample creation as an optimization problem. Optimizing data summaries based on query patterns has been widely studied in the database community as well [30, 54]. In the next section, we marry the two approaches to create an optimal set of samples within a fixed storage budget, i.e., given a collection of past QCS and their historical frequencies, we choose a collection of multi-dimensional stratified

samples with total storage costs below some user configurable storage threshold. These samples are designed to efficiently answer queries with the same QCSs as past queries, and to provide good coverage for future queries over similar QCS. If the distribution of QCSs is stable over time, our approach creates samples that are neither over- nor under-specialized for the query workload. We show that in real-world workloads from Facebook Inc. and Conviva Inc., stratified samples built using historical patterns of QCS usage continue to perform well for future queries.

2.3 Sample Creation

BlinkDB creates a set of samples to accurately and quickly answer queries. In this section, we describe the sample creation process in detail. First, we discuss the creation of a stratified sample on a given set of columns. We show how the query accuracy and response time depends on the availability of stratified samples for that query, and evaluate the storage requirements of our stratified sampling strategy for various data distributions. Stratified samples are useful, but carry storage costs, so we can only build a limited number of them. Next, we formulate and solve an optimization problem to decide on the sets of columns on which we build samples.

Stratified Samples

In this section, we describe our techniques for constructing a sample to target queries using a given QCS. Table 2.1 contains the notation used in the rest of this section.

Queries that do not filter or group data (for example, a SUM over an entire table) often produce accurate answers when run on uniform samples. However, uniform sampling often does not work well for a queries on filtered or grouped subsets of the table. When members of a particular subset are rare, a larger sample will be required to produce high-confidence estimates on that subset. A uniform sample may not contain any members of the subset at all, leading to a missing row in the final output of the query. The standard approach to solving this problem is *stratified sampling* [55], which ensures that rare subgroups are sufficiently represented. Next, we describe the use of stratified sampling in BlinkDB.

Optimizing a stratified sample for a single query

First, consider the smaller problem of optimizing a stratified sample for a single query. We are given a query Q specifying a table T, a QCS ϕ, and either a response time bound t or an error bound e. A time bound t determines the maximum (and optimal) sample size on which we can operate, n. Similarly, given an error bound e, it is possible to calculate the minimum sample size that will satisfy the error bound, and any larger sample would be suboptimal because it would take longer than necessary. In general n is monotonically increasing in t

Notation	Description
T	fact (original) table
Q	a query
t	a time bound for query Q
e	an error bound for query Q
n	the estimated number of rows that can be accessed in time t
ϕ	the QCS for Q, a set of columns in T
x	a $\|\phi\|$-tuple of values for a column set ϕ, for example (`Berkeley, CA`) for ϕ =(City, State)
$D(\phi)$	the set of all unique x-values for ϕ in T
T_x, S_x	the rows in T (or a subset $S \subseteq T$) having the values x on ϕ (ϕ is implicit)
$S(\phi, K)$	stratified sample associated with ϕ, where frequency of every group x in ϕ is capped by K
$\Delta(\phi, M)$	the number of groups in T under ϕ having size less than M — a measure of *sparsity* of T

Table 2.1: Notation used in Chapter 2

(or monotonically decreasing in e) but will also depend on Q and on the resources available in the cluster to process Q. We will show later in Chapter 3 how we estimate n at runtime using an *Error-Latency Profile*.

Among the rows in T, let $D(\phi)$ be the set of unique values x on the columns in ϕ. For each value x there is a set of rows in T having that value, $T_x = \{r : r \in T$ and r takes values x on columns $\phi\}$. We will say that there are $|D(\phi)|$ "groups" T_x of rows in T under ϕ. We would like to compute an aggregate value for each T_x (for example, a `SUM`). Since that is expensive, instead we will choose a sample $S \subseteq T$ with $|S| = n$ rows. For each group T_x there is a corresponding sample group $S_x \subseteq S$ that is a subset of T_x, which will be used instead of T_x to calculate an aggregate. The aggregate calculation for each S_x will be subject to error that will depend on its size. The best sampling strategy will minimize some measure of the expected error of the aggregate across all the S_x, such as the worst expected error or the average expected error.

A standard approach is *uniform sampling* — sampling n rows from T with equal probability. It is important to understand why this is an imperfect solution for queries that compute aggregates on groups. A uniform random sample allocates a random number of rows to each group. The size of sample group S_x has a hypergeometric distribution with n draws, population size $|T|$, and $|T_x|$ possibilities for the group to be drawn. The expected size of S_x is $n\frac{|T_x|}{|T|}$, which is proportional to $|T_x|$. For small $|T_x|$, there is a chance that $|S_x|$

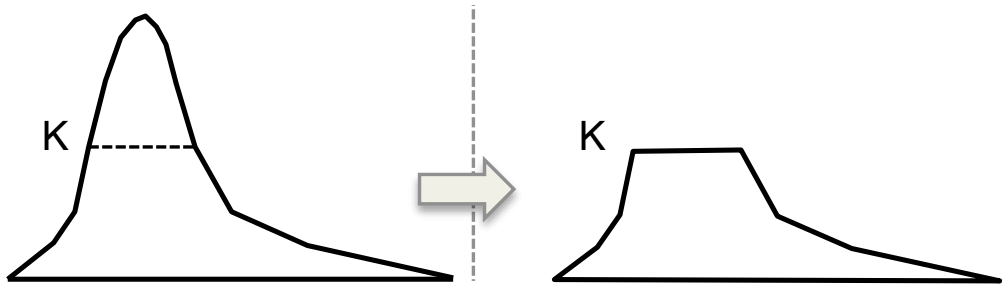


Figure 2.5: Example of a stratified sample associated with a set of columns, ϕ.

is very small or even zero, so the uniform sampling scheme can miss some groups just by chance. There are 2 things going wrong:

1. The sample size assigned to a group depends on its size in T. If we care about the error of each aggregate equally, it is not clear why we should assign more samples to S_x just because $|T_x|$ is larger.

2. Choosing sample sizes at random introduces the possibility of missing or severely under-representing groups. The probability of missing a large group is vanishingly small, but the probability of missing a small group is substantial.

This problem has been studied before. Briefly, since error decreases at a decreasing rate as sample size increases, the best choice simply assigns equal sample size to each groups. In addition, the assignment of sample sizes is deterministic, not random. A detailed proof is given by Acharya et al. [2]. This leads to the following algorithm for sample selection:

1. **Compute group counts:** To each $x \in x_0, ..., x_{|D(\phi)|-1}$, assign a count, forming a $|D(\phi)|$-vector of counts N_n^*. Compute N_n^* as follows: Let $N(n') = (\min(\lfloor \frac{n'}{|D(\phi)|} \rfloor, |T_{x_0}|), \min(\lfloor \frac{n'}{|D(\phi)|} \rfloor, |T_{x_1}|), ...)$, the optimal count-vector for a total sample size n'. Then choose $N_n^* = N(\max\{n' : ||N(n')||_1 \leq n\})$. In words, our samples cap the count of each group at some value $\lfloor \frac{n'}{|D(\phi)|} \rfloor$. In the future we will use the name K for the cap size $\lfloor \frac{n'}{|D(\phi)|} \rfloor$.

2. **Take samples:** For each x, sample N_{nx}^* rows uniformly at random without replacement from T_x, forming the sample S_x. Note that when $|T_x| = N_{nx}^*$, our sample includes all the rows of T_x, and there will be no sampling error for that group.

The entire sample $S(\phi, K)$ is the disjoint union of the S_x. Since a stratified sample on ϕ is completely determined by the group-size cap K, we henceforth denote a sample by $S(\phi, K)$

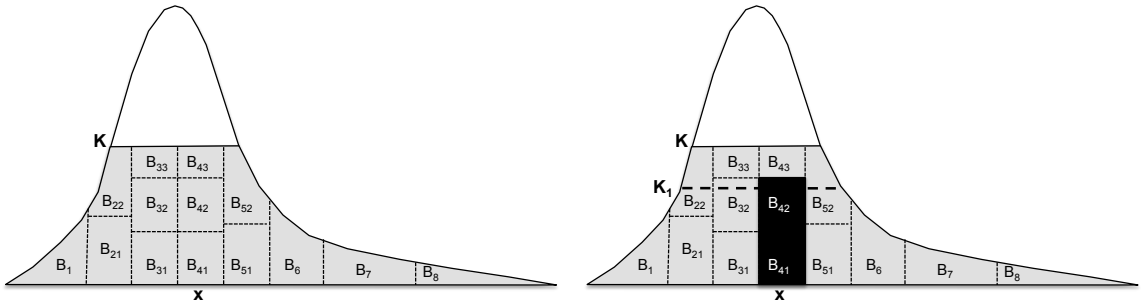


Figure 2.6: Possible storage layout for stratified sample $S(\phi, K)$.

or simply S when there is no ambiguity. K determines the size and therefore the statistical properties of a stratified sample for each group.

For example, consider query Q grouping by QCS ϕ, and assume we use $S(\phi, K)$ to answer Q. For each value x on ϕ, if $|T_x| \leq K$, the sample contains all rows from the original table, so we can provide an exact answer for this group. On the other hand, if $|T_x| > K$, we answer Q based on K random rows in the original table. For the basic aggregate operators `AVG`, `SUM`, `COUNT`, and `QUANTILE`, K directly determines the error of Q's result. In particular, these aggregate operators have standard error inversely proportional to $\sqrt{K}$ [55].

Optimizing a set of stratified samples for all queries sharing a QCS

Now we turn to the question of creating samples for a set of queries that share a QCS ϕ but have different values of n. Recall that n, the number of rows we read to satisfy a query, will vary according to user-specified error or time bounds. A `WHERE` query may also select only a subset of groups, which allows the system to read more rows for each group that is actually selected. So in general we want access to a family of stratified samples (S_n), one for each possible value of n.

Fortunately, there is a simple method that requires maintaining only a single sample for the whole family (S_n). According to our sampling strategy, for a single value of n, the size of the sample for each group is deterministic and is monotonically increasing in n. In addition, it is not necessary that the samples in the family be selected independently. So given any sample $S_{n^{max}}$, for any $n \leq n^{max}$ there is an $S_n \subseteq S_{n^{max}}$ that is an optimal sample for n in the sense of the previous section. Our sample storage technique, described next, allows such subsets to be identified at runtime.

The rows of stratified sample $S(\phi, K)$ are stored sequentially according to the order of columns in ϕ. Fig. 2.6 shows an example of storage layout for $S(\phi, K)$. B_{ij} denotes a data

block in the underlying file system, e.g., HDFS. Records corresponding to consecutive values in ϕ are stored in the same block, e.g., B_1. If the records corresponding to a popular value do not all fit in one block, they are spread across several contiguous blocks e.g., blocks B_{41}, B_{42} and B_{43} contain rows from S_x. Storing consecutive records contiguously on the disk significantly improves the execution times or range of the queries on the set of columns ϕ.

When S_x is spread over multiple blocks, each block contains a randomly ordered random subset from S_x, and, by extension, from the original table. This makes it possible to efficiently run queries on smaller samples. Assume a query Q, that needs to read n rows in total to satisfy its error bounds or time execution constraints. Let n_x be the number of rows read from S_x to compute the answer. (Note $n_x \leq \max\{K, |T_x|\}$ and $\sum_{x \in D(\phi), x \text{ selected by } Q} n_x = n$.) Since the rows are distributed randomly among the blocks, it is enough for Q to read any subset of blocks comprising S_x, as long as these blocks contain at least n_x records. Fig. 2.6(b) shows an example where Q reads only blocks B_{41} and B_{42}, as these blocks contain enough records to compute the required answer.

Storage overhead. An important consideration is the overhead of maintaining these samples, especially for heavy-tailed distributions with many rare groups. Consider a table with 1 billion tuples and a column set with a Zipf distribution with an exponent of 1.5. Then, it turns out that the storage required by sample $S(\phi, K)$ is only 2.4% of the original table for $K = 10^4$, 5.2% for $K = 10^5$, and 11.4% for $K = 10^6$.

These results are consistent with real-world data from Conviva Inc., where for $K = 10^5$, the overhead incurred for a sample on popular columns like city, customer, autonomous system number (ASN) is less than 10%.

Optimization Framework

We now describe the optimization framework to select subsets of columns on which to build sample families. Unlike prior work which focuses on single-column stratified samples [17] or on a single multi-dimensional (i.e., multi-column) stratified sample [2], BlinkDB creates several multi-dimensional stratified samples. As described above, each stratified sample can potentially be used at runtime to improve query accuracy and latency, especially when the original table contains small groups for a particular column set. However, each stored sample has a storage cost equal to its size, and the number of potential samples is exponential in the number of columns. As a result, we need to be careful in choosing the set of column-sets on which to build stratified samples. We formulate the trade-off between storage cost and query accuracy/performance as an optimization problem, described next.

Problem Formulation

The optimization problem takes three factors into account in determining the sets of columns on which stratified samples should be built: the *"sparsity"* of the data, *workload characteristics*, and the *storage cost of samples*.

Sparsity of the data. A stratified sample on ϕ is useful when the original table T contains many small groups under ϕ. Consider a QCS ϕ in table T. Recall that $D(\phi)$ denotes the set of all distinct values on columns ϕ in rows of T. We define a "sparsity" function $\Delta(\phi, M)$ as the number of groups whose size in T is less than some number M[4]:

$$\Delta(\phi, M) = |\{x \in D(\phi) : |T_x| < M\}|$$

Workload. A stratified sample is only useful when it is beneficial to actual queries. Under our model for queries, a query has a QCS q_j with some (unknown) probability p_j - that is, QCSs are drawn from a Multinomial $(p_1, p_2, ...)$ distribution. The best estimate of p_j is simply the frequency of queries with QCS q_j in past queries.

Storage cost. Storage is the main constraint against building too many stratified samples, and against building stratified samples on large column sets that produce too many groups. Therefore, we must compute the storage cost of potential samples and constrain total storage. To simplify the formulation, we assume a single value of K for all samples; a sample family ϕ either receives no samples or a full sample with K elements of T_x for each $x \in D(\phi)$. $|S(\phi, K)|$ is the storage cost (in rows) of building a stratified sample on a set of columns ϕ.

Given these three factors defined above, we now introduce our optimization formulation. Let the overall storage capacity budget (again in rows) be $\mathbb{C}$. Our goal is to select β column sets from among m possible QCSs, say $\phi_{i_1}, \cdots, \phi_{i_\beta}$, which can best answer our queries, while satisfying:

$$\sum_{k=1}^{\beta} |S(\phi_{i_k}, K)| \leq \mathbb{C}$$

Specifically, in BlinkDB, we maximize the following mixed integer linear program (MILP) in which j indexes over all queries and i indexes over all possible column sets:

$$G = \sum_j p_j \cdot y_j \cdot \Delta(q_j, M) \tag{2.1}$$

[4]Appropriate values for M will be discussed later in this section. Alternatively, one could plug in different notions of sparsity of a distribution in our formulation.

subject to

$$\sum_{i=1}^{m} |S(\phi_i, K)| \cdot z_i \leq \mathbb{C} \tag{2.2}$$

and

$$\forall\ j: \quad y_j \leq \max_{i:\phi_i \subseteq q_j \cup i:\phi_i \supset q_j} (z_i \min 1, \frac{|D(\phi_i)|}{|D(q_j)|}) \tag{2.3}$$

where $0 \leq y_j \leq 1$ and $z_i \in \{0, 1\}$ are variables.

Here, z_i is a binary variable determining whether a sample family should be built or not, i.e., when $z_i = 1$, we build a sample family on ϕ_i; otherwise, when $z_i = 0$, we do not.

The goal function (2.1) aims to maximize the weighted sum of the coverage of the QCSs of the queries, q_j. If we create a stratified sample $S(\phi_i, K)$, the coverage of this sample for q_j is defined as the probability that a given value x of columns q_j is also present among the rows of $S(\phi_i, K)$. If $\phi_i \supseteq q_j$, then q_j is covered exactly, but $\phi_i \subset q_j$ can also be useful by *partially covering* q_j. At runtime, if no stratified sample is available that exactly covers the QCS for a query, a partially-covering QCS may be used instead. In particular, the uniform sample is a degenerate case with $\phi_i = \emptyset$; it is useful for many queries but less useful than more targeted stratified samples.

Since the coverage probability is hard to compute in practice, in this thesis we approximate it by y_j, which is determined by constraint (2.3). The y_j value is in $[0, 1]$, with 0 meaning no coverage, and 1 meaning full coverage. The intuition behind (2.3) is that when we build a stratified sample on a subset of columns $\phi_i \subseteq q_j$, i.e., when $z_i = 1$, we have partially covered q_j, too. We compute this coverage as the ratio of the number of unique values between the two sets, i.e., $|D(\phi_i)|/|D(q_j)|$. When $\phi_i \subset q_j$, this ratio, and the true coverage value, is at most 1. When $\phi_i = q_j$, the number of unique values in ϕ_i and q_j are the same, we are guaranteed to see all the unique values of q_j in the stratified sample over ϕ_i and therefore the coverage will be 1. When $\phi_i \supset q_j$, the coverage is also 1, so we cap the ratio $|D(\phi_i)|/|D(q_j)|$ at 1.

Finally, we need to weigh the coverage of each set of columns by their importance: a set of columns q_j is more important to cover when: (i) it appears in more queries, which is represented by p_j, or (ii) when there are more small groups under q_j, which is represented by $\Delta(q_j, M)$. Thus, the best solution is when we maximize the sum of $p_j \cdot y_j \cdot \Delta(q_j, M)$ for all QCSs, as captured by our goal function (2.1).

The size of this optimization problem increases exponentially with the number of columns in T, which looks worrying. However, it is possible to solve these problems in practice by applying some simple optimizations, like considering only column sets that actually occurred in the past queries, or eliminating column sets that are unrealistically large.

Finally, we must return to two important constants we have left in our formulation, M and K. In practice we set $M = K = 100000$. Our experimental results in §2.4 show that the system performs quite well on the datasets we consider using these parameter values.

2.4 Evaluation

In this section, we evaluate BlinkDB's sampling strategy on a 100 node EC2 cluster using a workload from Conviva Inc. [36] and the well-known TPC-H benchmark [75]. First, we compare BlinkDB to query execution on full-sized datasets to demonstrate how even a small trade-off in the accuracy of final answers can result in orders-of-magnitude improvements in query response times. Second, we evaluate the accuracy and convergence properties of our optimal multi-dimensional stratified-sampling approach against both random sampling and single-column stratified-sampling approaches.

Evaluation Setting

The Conviva and the TPC-H datasets were 17 TB and 1 TB (i.e., a scale factor of 1000) in size, respectively, and were both stored across 100 Amazon EC2 extra large instances (each with 8 CPU cores (2.66 GHz), 68.4 GB of RAM, and 800 GB of disk). The cluster was configured to utilize 75 TB of distributed disk storage and 6 TB of distributed RAM cache.

Conviva Workload. The Conviva data represents information about video streams viewed by Internet users. We use query traces from their SQL-based ad-hoc querying system which is used for problem diagnosis and data analytics on a log of media accesses by Conviva users. These access logs are 1.7 TB in size and constitute a small fraction of data collected across 30 days. Based on their underlying data distribution, we generated a 17 TB dataset for our experiments and partitioned it across 100 nodes. The data consists of a single large *fact* table with 104 columns, such as `customer ID, city, media URL, genre, date, time, user OS, browser type, request response time`, etc. The 17 TB dataset has about 5.5 billion rows and shares all the key characteristics of real-world production workloads observed at Facebook Inc. and Microsoft Corp. [7].

The raw query log consists of 19,296 queries, from which we selected different subsets for each of our experiments. We ran our optimization function on a sample of about 200 queries representing 42 query column sets (with 1 to 13 columns in each set). We repeated the experiments with different storage budgets for the stratified samples– 50%, 100%, and

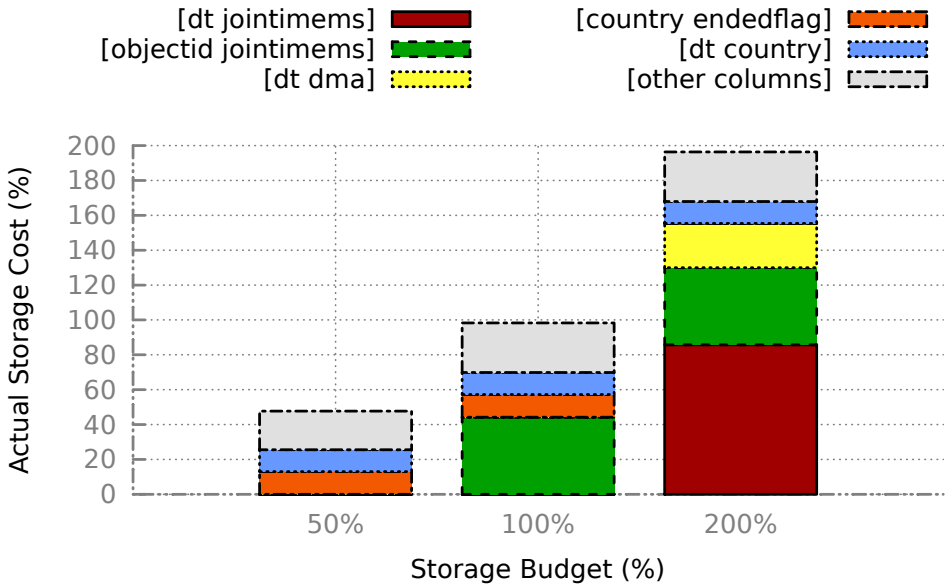
200%[5]. A storage budget of $s\%$ indicates that the cumulative size of all the samples will not exceed $\frac{s}{100}$ times the original data. So, for example, a budget of 100% indicates that the total size of all the samples should be less than or equal to the original data. Fig. 2.7a shows the set of samples that were selected by our optimization problem for the storage budgets of $50\%, 100\%$ and 200% respectively, along with their cumulative storage costs. Note that each stratified sample has a different size due to variable number of distinct keys in the table. For these samples, the value of K for stratified sampling is set to $100,000$.

TPC-H Workload. We also ran a smaller number of experiments using the TPC-H workload to demonstrate the generality of our results, with respect to a standard benchmark. All the TPC-H experiments ran on the same 100 node cluster, on 1 TB of data (i.e., a scale factor of 1000). The 22 benchmark queries in TPC-H were mapped to 6 unique query column sets. Fig. 2.7b shows the set of sample selected by our optimization problem for the storage budgets of $50\%, 100\%$ and 200%, along with their cumulative storage costs. Unless otherwise specified, all the experiments in this chapter are done with a 50% additional storage budget (i.e., samples could use additional storage of up to 50% of the original data size).

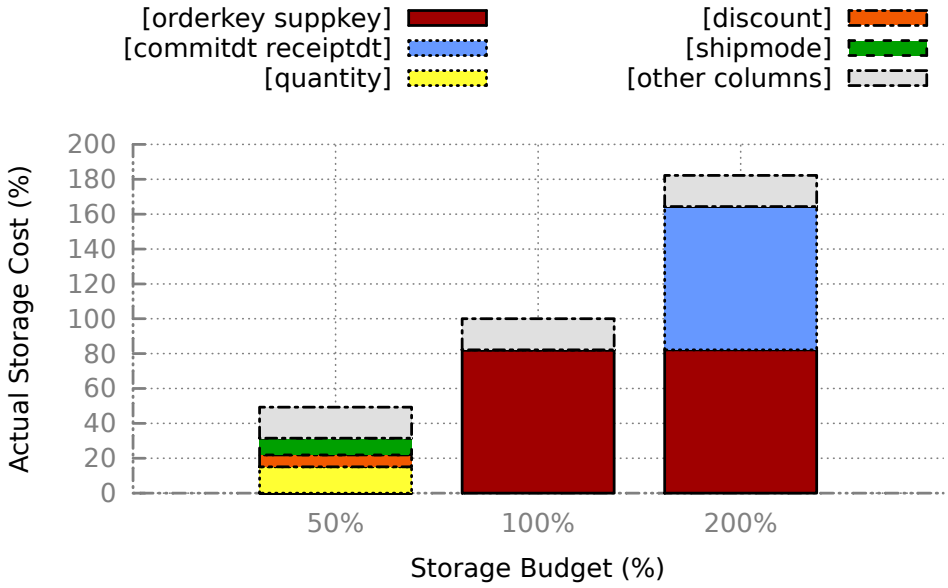
BlinkDB's Sampling vs. No Sampling

We first compare the performance of BlinkDB versus frameworks that execute queries on complete data. In this experiment, we ran on two subsets of the Conviva data, with 7.5 TB and 2.5 TB respectively, spread across 100 machines. We chose these two subsets to demonstrate some key aspects of the interaction between data-parallel frameworks and modern clusters with high-memory servers. While the smaller 2.5 TB dataset can be be completely cached in memory, datasets larger than 6 TB in size have to be (at least partially) spilled to disk. To demonstrate the significance of sampling even for the simplest analytical queries, we ran a simple query that computed `average` of user session times with a filtering predicate on the date column (dt) and a `GROUP BY` on the *city* column. We compared the response time of the full (accurate) execution of this query on Hive [74] on Hadoop MapReduce [15], Hive on Spark (called Shark [81]) – both with and without caching, against its (approximate) execution on BlinkDB with a 1% error bound for each `GROUP BY` key at 95% confidence. We ran this query on both data sizes (i.e., corresponding to 5 and 15 days worth of logs, respectively) on the aforementioned 100-node cluster. We repeated each query 10 times, and report the average response time in Figure 2.8. Note that the Y axis is log scale. In all cases, BlinkDB significantly outperforms its counterparts (by a factor of $10 - 200\times$), because it is able to read far less data to compute a fairly accurate answer. For both data sizes, BlinkDB returned the answers in a few seconds as compared to thousands of seconds for others. In the 2.5 TB run, Shark's caching capabilities help considerably, bringing the query runtime down to about 112 seconds. However, with 7.5 TB of data, a considerable portion of data is

[5]Please note that while individual stratified sample size is logarithmic in the size of data, there could be exponentially many possible samples to create, which in turn may require a considerable sampling budget.



(a) Biased Samples (Conviva)



(b) Biased Samples (TPC-H)

Figure 2.7: Fig. 2.7a and Fig. 2.7b show the relative sizes of the set of stratified sample(s) created for 50%, 100% and 200% storage budget on Conviva and TPC-H workloads respectively. The samples are identified by the sets of column(s) on which they are stratified on.

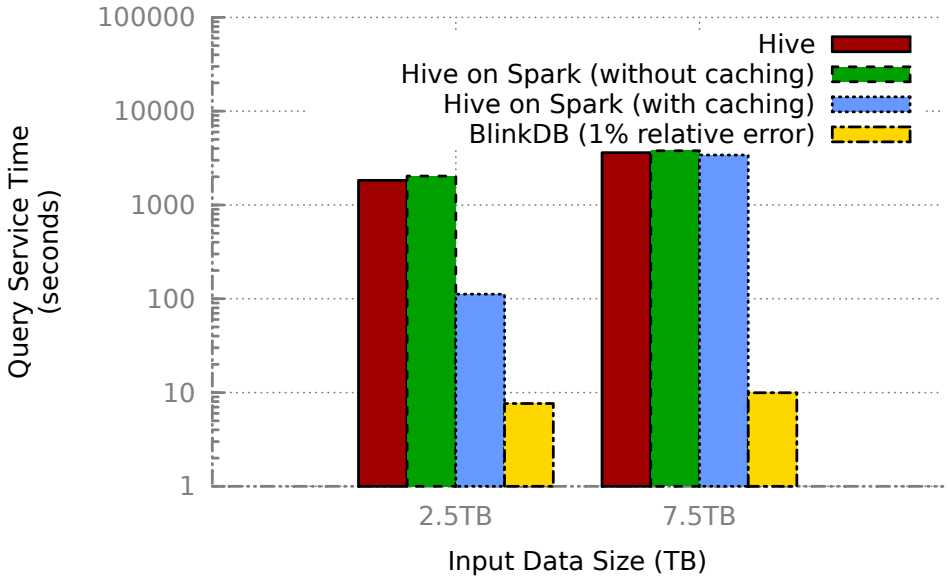


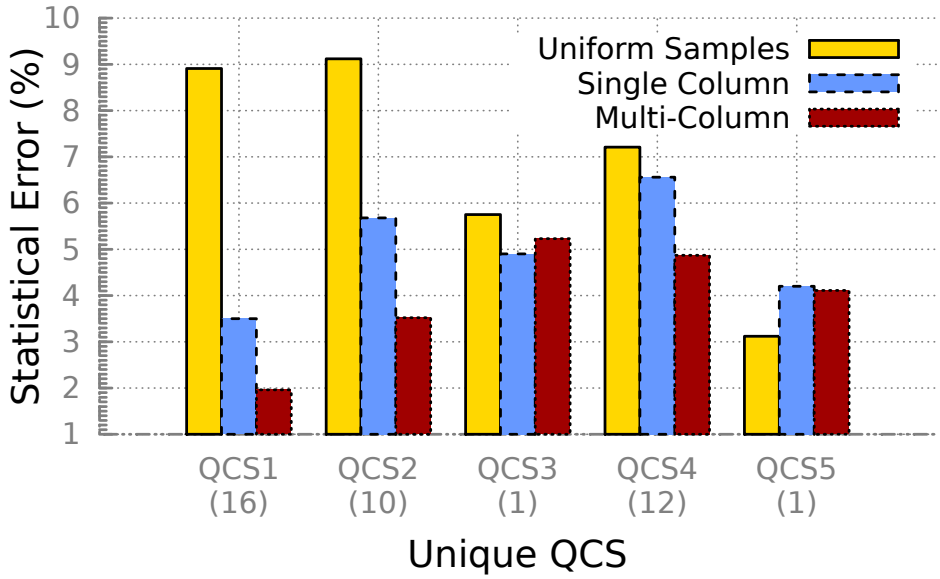
Figure 2.8: A comparison of response times (in log scale) incurred by Hive (on Hadoop), Shark (Hive on Spark) – both with and without input data caching, and BlinkDB, on simple aggregation.

spilled to disk and the overall query response time is considerably longer.

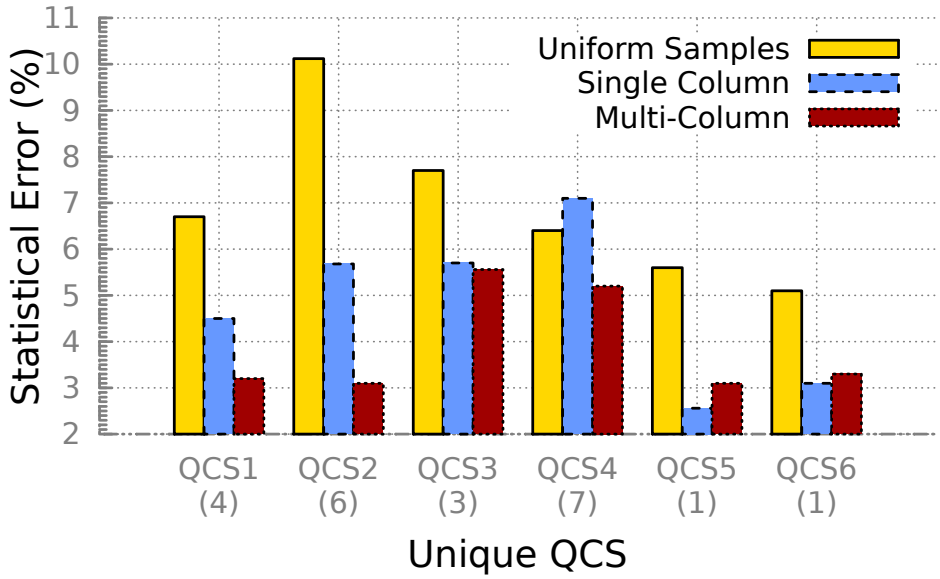
Multi-Dimensional Stratified Sampling

Next, we ran a set of experiments to evaluate the error and convergence properties of our optimal multi-dimensional stratified-sampling approach against both simple random sampling, and one-dimensional stratified sampling (i.e., stratified samples over a single column). For these experiments we constructed three sets of samples on both Conviva and TPC-H data with a 50% storage constraint:

1. **Uniform Samples**. A sample containing 50% of the entire data, chosen uniformly at random.
2. **Single-Dimensional Stratified Samples**. The column to stratify on was chosen using the same optimization framework, restricted so a sample is stratified on exactly one column.
3. **Multi-Dimensional Stratified Samples**. The sets of columns to stratify on were chosen using BlinkDB's optimization framework (§2.3), restricted so that samples could be stratified on no more than 3 columns (considering four or more column combinations caused our optimizer to take more than a minute to complete).



(a) Error Comparison (Conviva)



(b) Error Comparison (TPC-H)

Figure 2.9: Fig. 2.9a and Fig. 2.9b compare the average statistical error per QCS when running a query with fixed time budget of 10 seconds for various sets of samples.

Error Properties

In order to illustrate the advantages of our multi-dimensional stratified sampling strategy, we compared the average statistical error at 95% confidence while running a query for 10

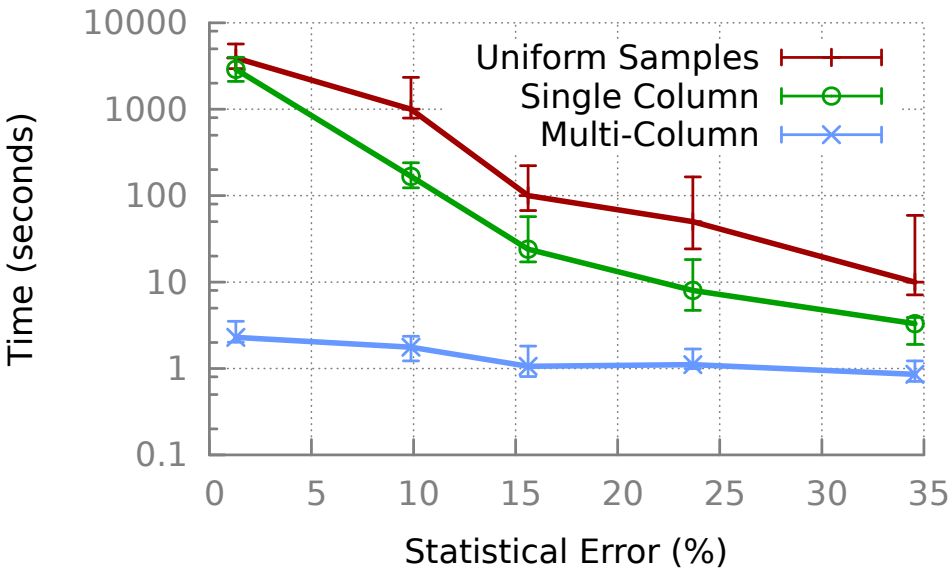


Figure 2.10: A comparison of the rates of error convergence with respect to time for various sets of samples.

seconds over the three sets of samples, all of which were constrained to be of the same size.

For our evaluation using Conviva's data we used a set of 40 of the most popular queries (with 5 unique QCSs) and 17 TB of uncompressed data on 100 nodes. We ran a similar set of experiments on the standard TPC-H queries (with 6 unique QCSs). The queries we chose were on the *lineitem* table, and were modified to conform with HiveQL syntax.

In Figures 2.9a, and 2.9b, we report the average statistical error in the results of each of these queries when they ran on the aforementioned sets of samples. The queries are binned according to the set(s) of columns in their `GROUP BY`, `WHERE` and `HAVING` clauses (i.e., their QCSs) and the numbers in brackets indicate the number of queries which lie in each bin. Based on the storage constraints, BlinkDB's optimization framework had samples stratified on QCS_1 and QCS_2 for Conviva data and samples stratified on QCS_1, QCS_2 and QCS_4 for TPC-H data. For common QCSs, multi-dimensional samples produce smaller statistical errors than either one-dimensional or random samples. The optimization framework attempts to minimize expected error, rather than per-query errors, and therefore for some specific QCS single-dimensional stratified samples behave better than multi-dimensional samples. Overall, however, our optimization framework significantly improves performance versus single column samples.

Convergence Properties

We also ran experiments to demonstrate the convergence properties of multi-dimensional stratified samples used by BlinkDB. We use the same set of three samples as §2.4, taken over 17 TB of Conviva data. Over this data, we ran multiple queries to calculate average session time for a particular ISP's customers in 5 US Cities and determined the latency for achieving a particular error bound with 95% confidence. Results from this experiment (Figure 2.10) show that error bars from running queries over multi-dimensional samples converge orders-of-magnitude faster than random sampling (i.e., Hadoop Online [35, 65]), and are significantly faster to converge than single-dimensional stratified samples.

2.5 Conclusion

In this chapter, we focused on the problem of creating and maintaining samples in BlinkDB. The sample creation module creates *stratified* samples on the most frequently used QCSs to ensure efficient execution for queries on rare values. By *stratified*, we mean that rare subgroups (e.g., `Galena, IL`) are over-represented relative to a uniformly random sample. This ensures that we can answer queries about any subgroup, regardless of its representation in the underlying data. We formulated the problem of sample creation as an optimization problem. Given a collection of past QCS and their historical frequencies, we choose a collection of stratified samples with total storage costs below some user configurable storage threshold. These samples are designed to efficiently answer queries with the same QCSs as past queries, and to provide good coverage for future queries over similar QCS. If the distribution of QCSs is stable over time, our approach creates samples that are neither over- nor under-specialized for the query workload. We showed that in real-world workloads from Facebook Inc. and Conviva Inc., QCSs do re-occur frequently and that stratified samples built using historical patterns of QCS usage continue to perform well for future queries. Next, we will focus on the problem of *runtime sample selection* in Chapter 3.

Chapter 3

Materialized Sample View Selection

In this chapter, we provide an overview of simple query execution in BlinkDB and present our approach for materialized sample view selection. Please note that in the rest of this chapter, we will use the term *runtime sample selection* to imply selecting pre-computed samples at runtime. Given a query Q, the goal is to select one (or more) sample(s) at *run-time* that meet the specified time or error constraints and then compute answers over them. Picking a sample involves selecting either the *uniform sample* or one of the *stratified samples* (none of which may stratify on exactly the QCS of Q), and then possibly executing the query on a subset of tuples from the selected sample. The selection of a sample (i.e., *uniform* or *stratified*) depends on the set of columns in Q's clauses, the selectivity of its selection predicates, and the data placement and distribution. In turn, the size of the sample subset on which we ultimately execute the query depends on Q's time/accuracy constraints, its computation complexity, the physical distribution of data in the cluster, and available cluster resources (i.e., empty slots) at runtime.

As with traditional query processing, accurately predicting the selectivity is hard, especially for complex `WHERE` and `GROUP BY` clauses. This problem is compounded by the fact that the underlying data distribution can change with the arrival of new data. Accurately estimating the query response time is even harder, especially when the query is executed in a distributed fashion. This is (in part) due to variations in machine load, network throughput, as well as a variety of non-deterministic (sometimes time-dependent) factors that can cause wide performance fluctuations.

Furthermore, maintaining a large number of samples (which are cached in memory to different extents), allows BlinkDB to generate many different query plans for the same query that may operate on different samples to satisfy the same error/response time constraints. In order to pick the best possible plan, BlinkDB's run-time dynamic sample selection strategy involves executing the query on a small sample (i.e., a *subsample*) of data of one or more samples and gathering statistics about the query's selectivity, complexity and the underly-

ing distribution of its inputs. Based on these results and the available resources, BlinkDB extrapolates the response time and relative error with respect to sample sizes to construct an *Error Latency Profile* (ELP) of the query for each sample, assuming different subset sizes. An ELP is a heuristic that enables quick evaluation of different query plans in BlinkDB to pick the one that can best satisfy a query's error/response time constraints. However, it should be noted that depending on the distribution of underlying data and the complexity of the query, such an estimate might not always be accurate, in which case BlinkDB may need to read additional data to meet the query's error/response time constraints.

In the rest of this chapter, we detail our approach to query execution, by first discussing our mechanism for selecting a set of appropriate samples (§3.1), and then picking an appropriate subset size from one of those samples by constructing the *Error Latency Profile* for the query (§3.2). Finally, we discuss how BlinkDB corrects the bias introduced by executing queries on stratified samples (§3.4).

3.1 Selecting the Optimal Stratified Sample

Choosing an appropriate sample for a query primarily depends on the set of columns q_j that occur in its `WHERE` and/or `GROUP BY` clauses and the physical distribution of data in the cluster (i.e., *disk vs. memory*). If BlinkDB finds one or more stratified samples on a set of columns ϕ_i such that $q_j \subseteq \phi_i$, we simply pick the ϕ_i with the smallest number of columns, and run the query on $S(\phi_i, K)$. However, if there is no stratified sample on a column set that is a superset of q_j, we run Q in parallel on *in-memory* subsets of all samples currently maintained by the system. Then, out of these samples we select those that have a high *selectivity* as compared to others, where *selectivity* is defined as the ratio of (i) the number of rows *selected* by Q, to (ii) the number of rows *read* by Q (i.e., number of rows in that sample). The intuition behind this choice is that the response time of Q increases with the number of rows it reads, while the error decreases with the number of rows Q's `WHERE/GROUP BY` clause selects.

3.2 Selecting the Optimal Sample Size

Once a set of samples is decided, BlinkDB needs to select a particular sample ϕ_i and pick an appropriately sized *subsample* in that sample based on the query's response time or error constraints. We accomplish this by constructing an ELP for the query. The ELP characterizes the rate at which the error decreases (and the query response time increases) with increasing sample sizes, and is built simply by running the query on smaller samples to estimate the selectivity and project latency and error for larger samples. For a distributed query, its runtime scales with sample size, with the scaling rate depending on the exact query structure (`JOINS, GROUP BYs` etc.), physical placement of its inputs and the underlying data

distribution [7]. The variation of error (or the variance of the estimator) primarily depends on the variance of the underlying data distribution and the actual number of tuples processed in the sample, which in turn depends on the selectivity of a query's predicates.

Error Profile: An error profile is created for all queries with error constraints. If Q specifies an error (e.g., standard deviation) constraint, the BlinkDB error profile tries to predict the size of the smallest sample that satisfies Q's error constraint. Variance and confidence intervals for aggregate functions are estimated using standard closed-form formulas from statistics [55]. For all standard SQL aggregates, the variance is proportional to $\sim 1/n$, and thus the standard deviation (or the statistical error) is proportional to $\sim 1/\sqrt{n}$, where n is the number of rows from a sample of size N that match Q's filter predicates. Using this notation. the *selectivity* s_q of the query is the ratio n/N.

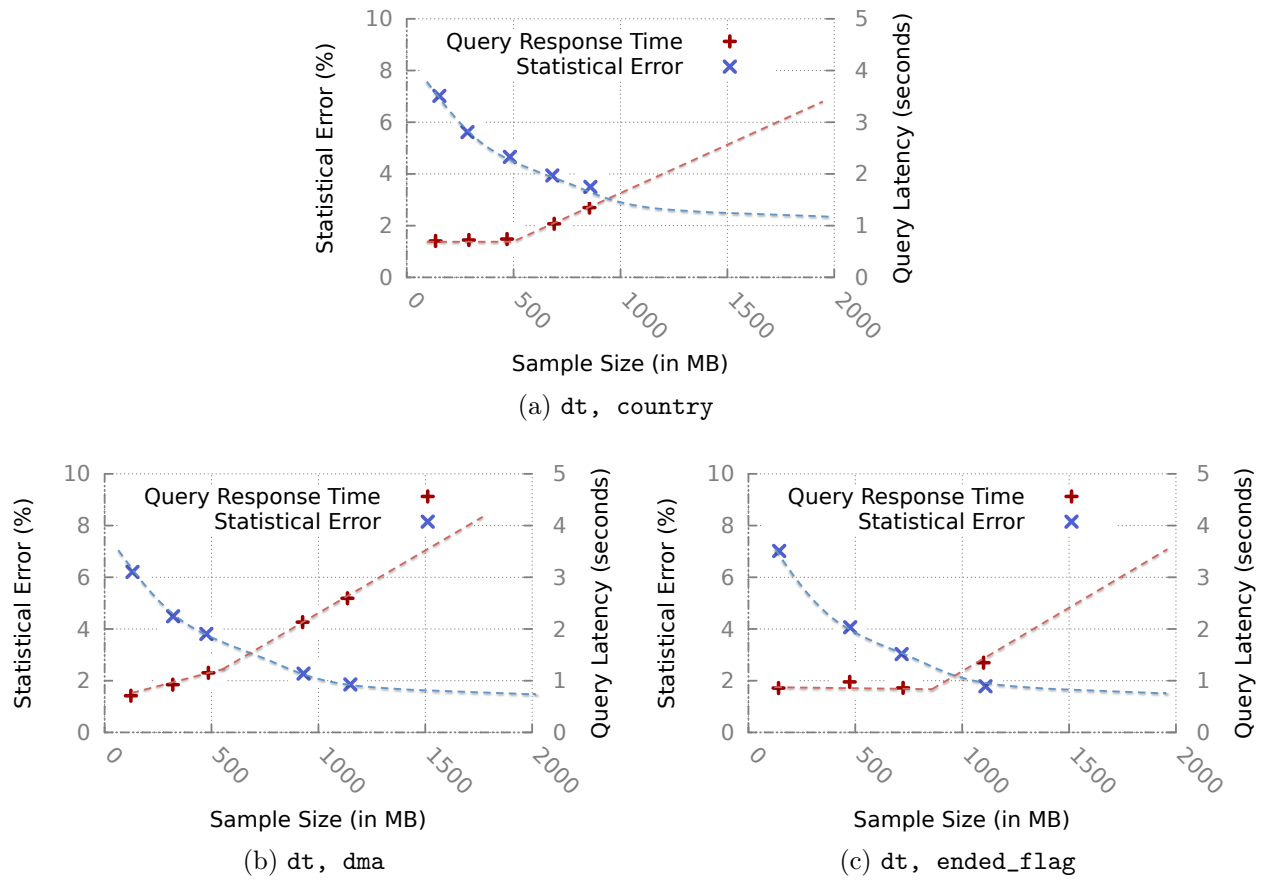
Let $n_{i,m}$ be the number of rows selected by Q when running on a subset m of the stratified sample, $S(\phi_i, K)$. Furthermore, BlinkDB estimates the query selectivity s_q, sample variance S_n (for `AVG/SUM`) and the input data distribution f (for `Quantiles`) by running the query on a number of small sample subsets. Using these parameter estimates, we calculate the number of rows $n = n_{i,m}$ required to meet Q's error constraints using standard closed form statistical error estimates [55]. Then, we run Q on $S(\phi_i, K)$ until it reads n rows.

Latency Profile: Similarly, a latency profile is created for all queries with response time constraints. If Q specifies a response time constraint, we select the sample on which to run Q the same way as above. Again, let $S(\phi_i, K)$ be the selected sample, and let n be the maximum number of rows that Q can read without exceeding its response time constraint. Then we simply run Q until reading n rows from $S(\phi_i, K)$.

The value of n depends on the physical placement of input data (disk vs. memory), the query structure and complexity, and the degree of parallelism (or the resources available to the query). As a simplification, BlinkDB simply predicts n by assuming that latency scales linearly with input size, as is commonly observed with a majority of I/O bounded queries in parallel distributed execution environments [11, 83]. To avoid non-linearities that may arise when running on very small in-memory samples, BlinkDB runs a few smaller samples until performance seems to grow linearly and then estimates the appropriate linear scaling constants (i.e., *data processing rate(s), disk/memory I/O rates etc.*) for the model.

3.3 An Example

As an illustrative example consider a query that calculates the average session time for a (`city`, `state`) tuple "`Galena, IL`". For the purposes of this example, the system has three stratified samples, one biased on `date` and `country`, one biased on `date` and the designated



(a) `dt, country`

(b) `dt, dma`

(c) `dt, ended_flag`

Figure 3.1: Error Latency Profiles for a variety of samples when executing a query to calculate average session time in `Galena`. (a) Shows the ELP for a sample biased on `date` and `country`, (b) is the ELP for a sample biased on `date` and designated media area (`dma`), and (c) is the ELP for a sample biased on `date` and the `ended_flag`. Please note that these ELPs differ primarily due to the nature and placement of the underlying samples.

media area (`dam`) for a video, and the last one biased on `date` and `ended_flag`. In this case it is not obvious which of these three samples would be preferable for answering the query.

In this case, BlinkDB constructs an ELP for each of these samples as shown in Figure 3.1. For many queries it is possible that all of the samples can satisfy specified time or error bounds. For instance all three of the samples in our example can be used to answer this query with an error bound of under 4%. However it is clear from the ELP that the sample biased on `date` and `ended_flag` would take the shortest time to find an answer within the required error bounds (perhaps because the data for this sample is cached), and BlinkDB would hence execute the query on that sample.

URL	City	Browser	SessionTime
cnn.com	New York	Firefox	15
yahoo.com	New York	Firefox	20
google.com	Berkeley	Firefox	85
google.com	New York	Safari	82
bing.com	Cambridge	IE	22

Table 3.1: `Sessions` Table.

URL	City	Browser	SessionTime	SampleRate
yahoo.com	New York	Firefox	20	0.33
google.com	New York	Safari	82	1.0
bing.com	Cambridge	IE	22	1.0

Table 3.2: A sample of `Sessions` Table stratified on `Browser` column

3.4 Bias Correction

Running a query on a non-uniform sample introduces a certain amount of statistical bias in the final result since different groups are picked at different frequencies. In particular while all the tuples matching a rare subgroup would be included in the sample, more popular subgroups will only have a small fraction of values represented. To correct for this bias, BlinkDB keeps track of the effective sampling rate for each group associated with each sample in a hidden column as part of the sample table schema, and uses this to weight different subgroups to produce an unbiased result. To illustrate this further, consider the *Sessions* table, shown in Table 3.1, and the following query against this table.

```
SELECT City, SUM(SessionTime)
FROM Sessions
GROUP BY City
WITHIN 5 SECONDS
```

If we have a uniform sample of this table, estimating the query answer is straightforward. For instance, suppose we take a uniform sample with 40% of the rows of the original *Sessions* table. In this case, we simply scale the final sums of the session times by $1/0.4 = 2.5$ in order to produce an unbiased estimate of the true answer[1].

[1]Here we use the terms *biased* and *unbiased* in a statistical sense, meaning that although the estimate might vary from the actual answer, its *expected value* will be the same as the actual answer.

Using the same approach on a stratified sample may produce a biased estimate of the answer for this query. For instance, consider a stratified sample of the *Sessions* table on the *Browser* column, as shown in Table 3.2. Here, we have a cap value of $K = 1$, meaning we keep all rows whose *Browser* only appears once in the original *Sessions* table (e.g., *Safari* and *IE*), but when a browser has more than one row (i.e., *Firefox*), only one of its rows is chosen, uniformly at random. In this example we have to choose the row that corresponds to *Yahoo.com*. Here, we cannot simply scale the final sums of the session times because different values were sampled with different rates. Therefore, to produce unbiased answers, BlinkDB keeps track of the effective sampling rate applied to each row, e.g., in Table 3.2, this rate is 0.33 for *Firefox* row, while it is 1.0 for *Safari* and *IE* rows since they have not been sampled at all. Given these per-row sample rates, obtaining an unbiased estimates of the final answer is straightforward, e.g., in this case the sum of sessions times is estimated as $\frac{1}{0.33} \times 20 + \frac{1}{1} \times 82$ for *New York* and as $\frac{1}{1} \times 22$ for *Cambridge*. Note that here we will not produce any output for `Berkeley` (this would not happen if we had access to a stratified sample over *City*, for example). In general, the query processor in BlinkDB performs a similar correction when operating on stratified samples.

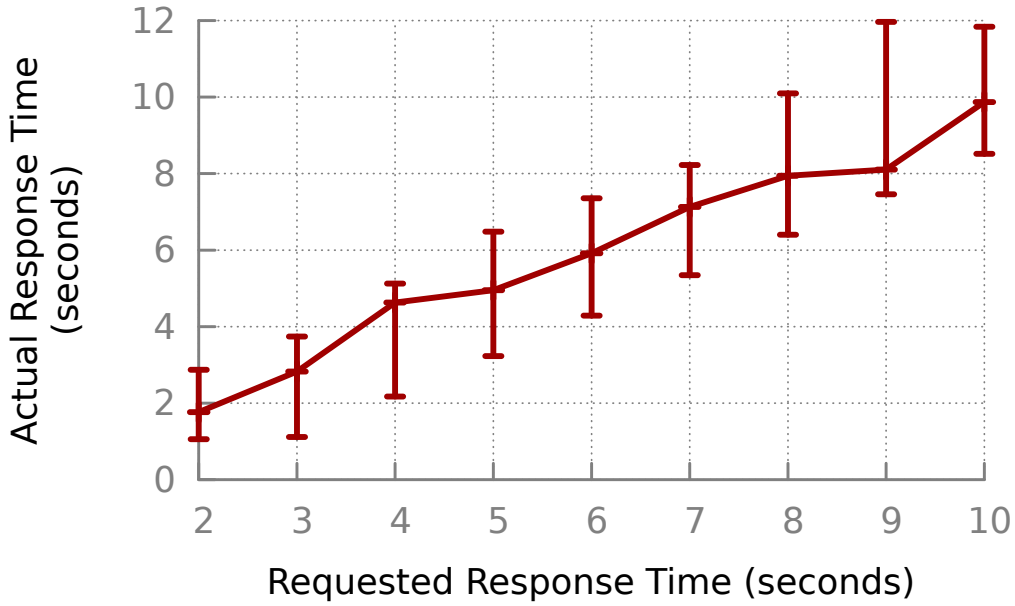
3.5 Evaluation

In this section, we evaluate BlinkDB's sample selection strategy on a 100 node EC2 cluster using a workload from Conviva Inc. [36] and the well-known TPC-H benchmark [75]. First, we evaluate BlinkDB's effectiveness at meeting different time/error bounds requested by the user. Second, we evaluate the scalability properties of BlinkDB as a function of cluster size.

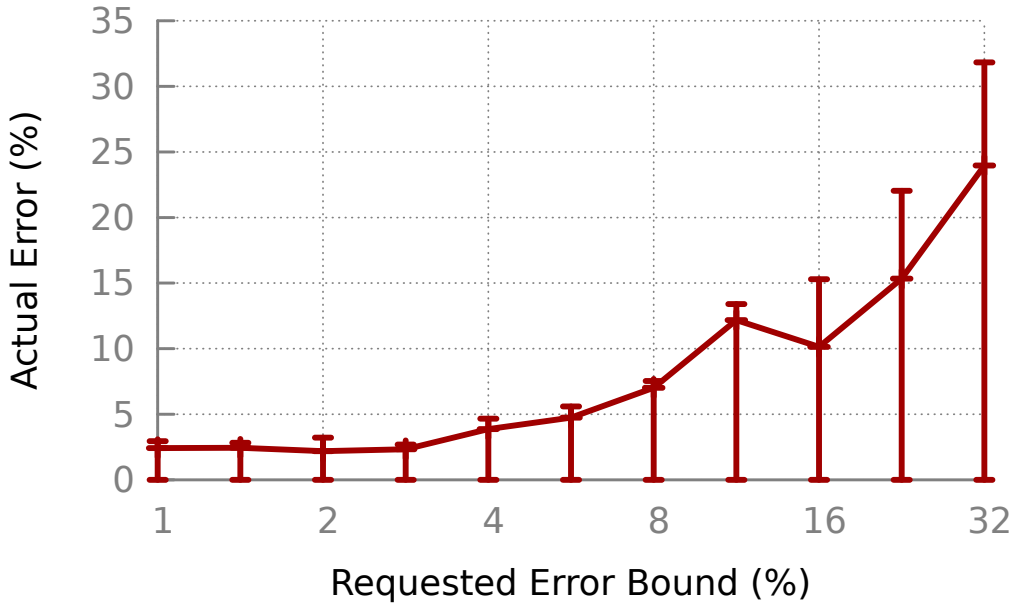
Evaluation Setting

We use the same set of queries and workloads that were used in Chapter 2. The Conviva and the TPC-H datasets were 17 TB and 1 TB (i.e., a scale factor of 1000) in size, respectively, and were both stored across 100 Amazon EC2 extra large instances (each with 8 CPU cores (2.66 GHz), 68.4 GB of RAM, and 800 GB of disk). The cluster was configured to utilize 75 TB of distributed disk storage and 6 TB of distributed RAM cache.

Conviva Workload. The Conviva data, as we mentioned, represents information about video streams viewed by Internet users. We use query traces from their SQL-based ad-hoc querying system which is used for problem diagnosis and data analytics on a log of media accesses by Conviva users. These access logs are 1.7 TB in size and constitute a small fraction of data collected across 30 days. Based on their underlying data distribution, we generated a 17 TB dataset for our experiments and partitioned it across 100 nodes. The data consists of a single large *fact* table with 104 columns, such as `customer ID, city, media URL, genre, date, time, user OS, browser type, request response time`, etc.



(a) Response Time Bounds



(b) Relative Error Bounds

Figure 3.2: Fig. 3.2a and Fig. 3.2b plot the actual vs. requested maximum response times and error bounds in BlinkDB.

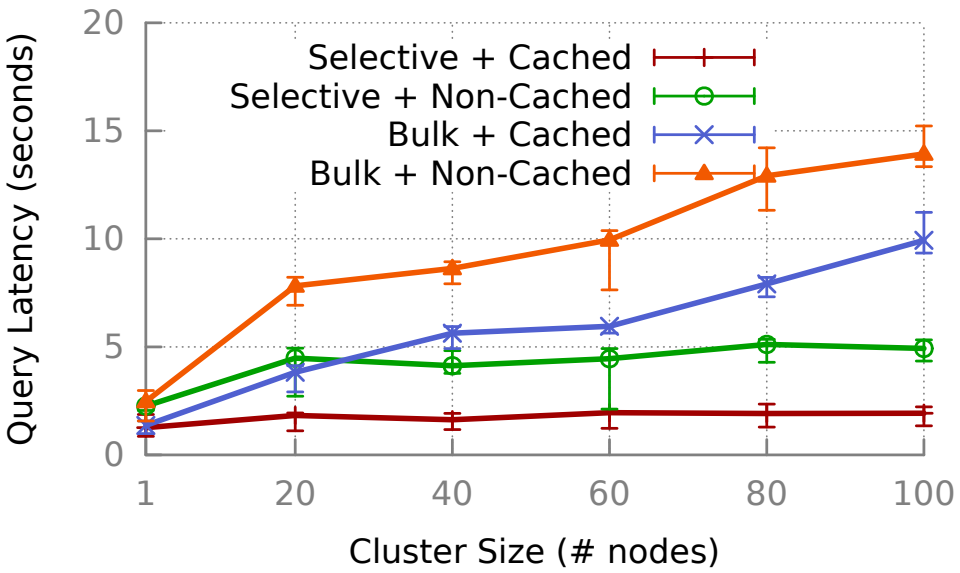


Figure 3.3: Query latency across different query workloads (with cached and non-cached samples) as a function of cluster size.

Time/Accuracy Guarantees

First, we evaluate BlinkDB's effectiveness at meeting different time/error bounds requested by the user. To test time-bounded queries, we picked a sample of 20 Conviva queries, and ran each of them 10 times, with a maximum time bound from 1 to 10 seconds. Figure 3.2a shows the results run on the same 17 TB data set, where each bar represents the minimum, maximum and average response times of the 20 queries, averaged over 10 runs. From these results we can see that BlinkDB is able to accurately select a sample to satisfy a target response time.

Figure 3.2b shows results from the same set of queries, also on the 17 TB data set, evaluating our ability to meet specified error constraints. In this case, we varied the requested maximum error bound from 2% to 32% . The bars again represent the minimum, maximum and average errors across different runs of the queries. Note that the measured error is almost always at or less than the requested error. However, as we increase the error bound, the measured error becomes closer to the bound. This is because at higher error rates the sample size is quite small and error bounds are wider.

Scaling Up

Finally, in order to evaluate the scalability properties of BlinkDB as a function of cluster size, we created 2 different sets of query workload suites consisting of 40 unique Conviva queries

each. The first set (marked as *selective*) consists of highly selective queries – i.e., those queries that only operate on a small fraction of input data. These queries occur frequently in production workloads and consist of one or more highly selective WHERE clauses. The second set (marked as *bulk*) consists of those queries that are intended to crunch huge amounts of data. While the former set's input is generally striped across a small number of machines, the latter set of queries generally runs on data stored on a large number of machines, incurring a higher communication cost. Figure 3.3 plots the query latency for each of these workloads as a function of cluster size. Each query operates on $100n$ GB of data (where n is the cluster size). So for a 10 node cluster, each query operates on 1 TB of data and for a 100 node cluster each query operates on around 10 TB of data. Further, for each workload suite, we evaluate the query latency for the case when the required samples are completely cached in RAM or when they are stored entirely on disk. Since in reality any sample will likely partially reside both on disk and in memory these results indicate the min/max latency bounds for any query.

3.6 Conclusion

In this chapter, we focussed on the problem of runtime materialized sample view selection. Based on a query's error/response time constraints, the sample selection module dynamically picks a sample on which to run the query. It does so by running the query on multiple smaller sub-samples (which could potentially be stratified across a range of dimensions) to quickly estimate query selectivity and choosing the best sample to satisfy specified response time and error bounds. We proposed an *Error-Latency Profile* heuristic to efficiently choose the sample that will best satisfy the user-specified error or time bounds and demonstrated BlinkDB provides bounded error and latency for a wide range of real-world SQL queries, and it is robust to variations in the query workload. Once we have an optimal sample, we will now shift our focus on ways of optimizing query execution in a parallel environment.

Chapter 4

Error Estimation

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

4.1 Approximate Query Processing (AQP)

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

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

The θ corresponding to this query is:

$$\theta(D) = \frac{1}{N} \sum_{t \in \sigma_{\text{City = 'NYC'}}(D)} t.Time \tag{4.1}$$

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

It is common for an analytical (a.k.a. OLAP) query to evaluate one or more such aggregate functions, each of which may output a set of values (due to the presence of `GROUP BY`). However, for the sake of simplicity, we assume that each query evaluates a single aggregate function that returns a single real number. When a query in our experimental dataset produces multiple results, we treat each result as a separate query. Additionally note that, while we focus on θs that encode analytical SQL queries, θ could instead correspond to a different aggregation function, like a MapReduce job in EARL [53].

When the size of the dataset ($|D|$) is too large or when the user prefers lower latency than the system can deliver, it is possible to save on I/O and computation by processing the query on less data, resorting to *approximation*. In particular, sampling has served as one of the most common and generic approaches to approximation of analytical queries [1, 5, 28, 35, 39, 47, 53, 66, 68]. The simplest form of sampling is *simple random sampling* with *plug-in estimation*. Instead of computing $\theta(D)$, this method identifies a random sample $S \subseteq D$ by sampling $n = |S| \leq |D|$ rows uniformly at random from D with replacement[1], and then returns the *sample estimate* $\theta(S)$ to the user[2]. Since $\theta(S)$ depends on the particular sample we identified, it is random, and we say that it is a draw from its *sampling distribution* $\text{Dist}(\theta[S])$. The random quantity $\varepsilon = \theta(S) - \theta(D)$ is the *sampling error*; it also has a probability distribution, which we denote by $\text{Dist}(\varepsilon)$.

4.2 An Overview of Error Estimation

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

Procedures for generating confidence intervals do not always work well, and we need a way to evaluate them. Unfortunately, to say that a procedure has correct coverage is not enough to pin down its usefulness. For example, a procedure can trivially achieve α coverage

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

[2] Another function $\hat{\theta}(S)$ may be used to estimate $\theta(D)$. For example, when θ is `SUM`, a reasonable choice of $\hat{\theta}$ would be the sample sum multiplied by a scaling factor $\frac{|D|}{|S|}$. The proper choice of $\hat{\theta}$ for a given θ is not the focus of this thesis and has been discussed in [59]. For simplicity of presentation we assume here that θ is given and that it appropriately handles scaling.

by returning $(-\infty, \infty)$ α of the time, and $\emptyset$ the rest of the time; obviously this procedure is not helpful in estimating error.

To resolve this technical problem, we choose to use *symmetric centered confidence intervals*. These do not have the unreasonable behavior of the above example, and further it is possible to evaluate them as estimates of a ground truth value: the interval centered around $\theta(D)$ that covers exactly the proportion α of the sampling distribution of $\theta(S)$, $\text{Dist}(\theta[S])$[3]. Though it is not really a confidence interval, since it is deterministic and always covers $\theta(D)$, in a slight abuse of terminology we call this ground truth value the *true confidence interval*. The procedure for generating symmetric centered confidence intervals from samples closely mimics the definition of the true value. Given a sample, we can estimate $\theta(D)$ as usual by $\theta(S)$. As we detail in the next section, various error estimation techniques can be used to estimate the sampling distribution $\text{Dist}(\theta[S])$. These two estimates ($\theta(S)$ and the estimate of its distribution) are in turn used to produce a confidence interval by finding an interval centered on $\theta(S)$ covering α of the estimated sampling distribution. This interval can be computed by finding the number a that satisfies $P([\theta(S) - a, \theta(S) + a]) = \alpha$, where P is the estimate of $\text{Dist}(\theta[S])$. We can evaluate such a confidence interval by merely comparing its width with that of the true confidence interval, i.e. by computing $\delta = \frac{\text{(true confidence interval width)} - \text{(estimated interval width)}}{\text{(true confidence interval width)}}$. If this quantity is much above or below zero for a particular query, we can declare that confidence interval estimation has failed in that case. Unlike coverage, this evaluation fully captures problems with error estimation. See [46] for a more detailed discussion of symmetric centered confidence intervals.

4.3 Estimating the Sampling Distribution

We now turn to the problem of estimating $\text{Dist}(\theta[S])$ using only a single sample S. There are three widely-used methods: the bootstrap [40, 85]; closed-form estimates based on normal approximations; and large deviation bounds [56]. It may be unclear why they could fail, and in order to aid intuition we provide a brief explanation of the methods.

Nonparametric Bootstrap

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

However, actually accessing D many times would defeat the purpose of sampling, namely, the need to use only a small amount of data. Instead, Efron's *nonparametric bootstrap* [40]

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

(or simply the *bootstrap*) uses the sample S in place of the dataset D, just as we used $\theta(S)$ as an estimate for $\theta(D)$ earlier. This means we take K "resamples" of size n (with replacement[4]) from S, which we denote S^i; compute $\theta(S^i)$ for each one; and take this "bootstrap resampling distribution" as our estimate of the sampling distribution. We make a further approximation by taking K to be merely a reasonably large number, like 100 (K can be tuned automatically [41]).

The substitution of S for D in the bootstrap does not always work well. Roughly, its accuracy depends on two things[5]:

1. **The sensitivity of $\boldsymbol{\theta}$ to outliers in $\boldsymbol{D}$**: If θ is sensitive to rare values and D contains such values, then the estimate can be poor (consider, for example, $\theta = \texttt{MAX}$).
2. **The sample size $\boldsymbol{n}$**: When the sensitivity condition is met, theory only tells us that the estimate is accurate in the limit as $n \to \infty$. The estimate can be poor even for moderately large n (for example $1,000,000$), and the theory gives no guidance on the requirements for n. It should be noted that the relationship between n and the accuracy of error bars is an exact analogue of the relationship between n and the size of the sampling error $|\varepsilon|$. It is therefore unlikely that the dependence on n will be avoided by finding a different method for computing confidence intervals; any reasonable method will suffer from some inaccuracy at small n and improve in accuracy as n grows.

In addition to the possibility of failure, the bootstrap involves a large amount of computation (K replications of the query on resamples), which can partially offset the benefits of sampling. Fig. 4.1 illustrates the use of the bootstrap in computing confidence intervals and its potential cost. This motivates the consideration of alternative methods.

Normal approximation and closed-form estimate of variance

This method approximates the sampling distribution by a normal distribution $N(\theta(S), \sigma^2)$ and estimates the variance σ^2 by a special closed-form function of the sample. We call this *closed-form estimation* for short. The normal approximation is justified by appealing to the central limit theorem. Like the bootstrap, this rests on assumptions of insensitivity of θ to outliers in D and on n being large. However, closed-form estimation replaces the brute-force computation of the bootstrap with estimation of the parameter σ^2 from S through careful manual study of θ. For example, a well-known estimate for σ^2 when $\theta(S)$ is `AVG` is

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

[5]There are two standard books by Van der Vaart ([77, 78]) that treat this in great detail.

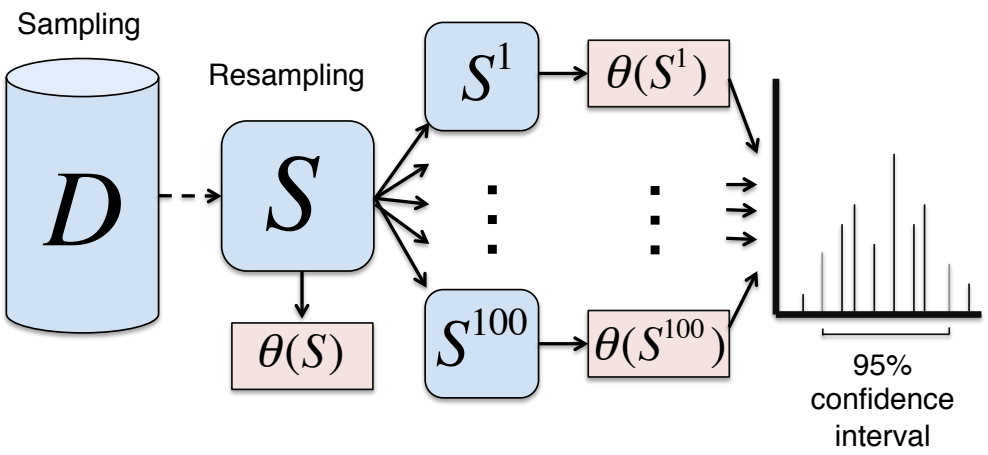


Figure 4.1: The computational pattern of bootstrap.

$s^2 = \frac{\text{Var}(S)}{n}$. Calculating $\text{Var}(S)$ is much cheaper than computing K bootstrap replicates, so in the case of `AVG`, closed-form estimation is appealing. The story is similar for several θs commonly used in SQL queries: `COUNT`, `SUM`, `AVG`, and `VARIANCE`. Other θs require more complicated estimates of σ^2, and in some cases, like `MIN`, `MAX`, and black-box *user defined functions* (UDFs), closed-form estimates are unknown. Therefore closed-form estimation is less general than the bootstrap. In our Facebook trace, 37.21% of queries are amenable to closed-form estimates. Later, in Fig. 6.4a we show the overhead of estimating the error using closed forms for a set of 100 randomly chosen queries that computed `COUNT`, `SUM`, `AVG`, or `VARIANCE` from our Facebook trace. When closed-form estimation is possible, it generally runs faster than the bootstrap.

Large Deviation Bounds

Large deviation bounds[6] [56] are a third technique for error estimation, used in several existing AQP systems including OLA[47, 45] and AQUA[1]. Rather than directly estimating the sampling distribution like the bootstrap and closed forms, this method bounds the *tails* of the sampling distribution; these bounds are sufficient to compute confidence intervals. The method relies on the direct computation of a quantity related to the "sensitivity to outliers" (for example, $|\max D - \min D|$ in the case of `SUM`) that, as previously mentioned, can ruin the accuracy of bootstrap- and closed-form-based error estimation. This sensitivity quantity depends on D and θ, so it must be precomputed for every θ and, like σ^2 in closed-form estimation, requires difficult manual analysis of θ. By making a worst-case assumption about the presence of outliers, large deviation bounds ensure that the resulting confidence intervals never have coverage less than α. That is, error bars based on large deviation

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

bounds will never be too small. However, this conservatism comes at a great cost: as we observe in Fig. 1.1, typically large deviation bounds instead produce confidence intervals much wider than the true confidence interval and with coverage much higher than α, making them extremely inefficient in practice.

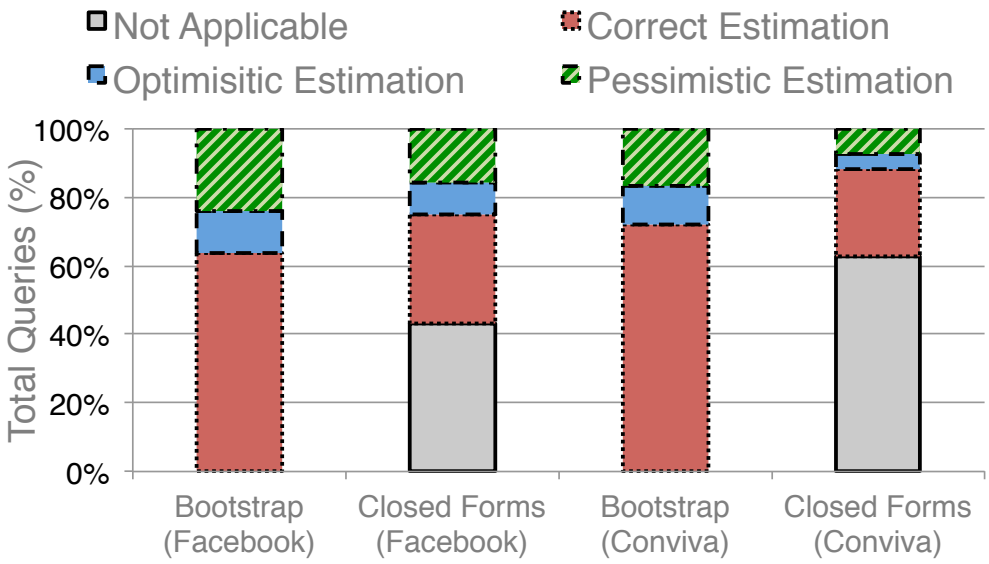
4.4 Problem: Estimation Fails

As noted in the previous section, all three estimation methods have modes of failure. We've already highlighted the pessimistic nature of large deviation bounds in Fig. 1.1, and in this section we evaluate the severity of these failure modes for the bootstrap and closed-form estimation. Recall that we can summarize the accuracy of a symmetric centered confidence interval by δ, the relative deviation of its width from that of the true interval. Ideally, we would like δ to be close to *zero*. If it is often positive and large, this means our procedure produced confidence intervals that are too large, and the intervals are typically larger than the true sampling error ε. In that case, we say that the procedure is *pessimistic*. (In Fig. 1.1 we saw an example– confidence intervals based on Hoeffding bounds suffer from extreme pessimism.) On the other hand, if δ is often much smaller than 0, then our procedure is producing confidence intervals that are misleadingly small, and we say that it is *optimistic*.

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

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

To test the usefulness of error estimation techniques on real OLAP queries, we analyzed a representative trace of $69,438$ Hive queries from Facebook constituting a week's worth of production queries during the 1^{st} week of Feb 2013 and a trace of $18,321$ Hive queries from Conviva Inc. [36] constituting a sample of a month's worth of production queries during Feb 2013. Among these, `MIN`, `COUNT`, `AVG`, `SUM`, and `MAX` were the most popular aggregate functions at Facebook constituting 33.35% , 24.67%, 12.20%, 10.11% and 2.87% of the total queries respectively. 11.01% of these queries consisted of one or more *UDFs* or *User-Defined*



	Not Applicable	Correct Estimation	Optimisitic Estimation	Pessimistic Estimation
Bootstrap (Facebook)	0%	63.84%	12.21%	23.95%
Closed Forms (Facebook)	43.22%	31.91%	9.12%	15.75%
Bootstrap (Conviva)	0%	72.24%	11.23%	16.53%
Closed Forms (Conviva)	62.79%	25.32%	4.32%	7.57%

Figure 4.2: Estimation Accuracy for bootstrap and closed-form based error estimation methods on real world Hive query workloads from Facebook (69, 438 queries) and Conviva (18, 321 queries).

Functions. In Conviva on the other hand, `AVG`, `COUNT`, `PERCENTILES`, and `MAX` were the most popular aggregate functions with a combined share of 32.3%. 42.07% of the Conviva queries had at least one UDF. While we are unable to disclose the exact set of queries that we used in our analysis due to the proprietary nature of the query workload and the underlying datasets, we have published a synthetic benchmark [10] that closely reflects the key characteristics of the Facebook and Conviva workloads presented in this chapter– both

in terms of the distribution of underlying data and the query workload. The input data set in this benchmark consists of a set of unstructured HTML documents and SQL tables that were generated using Intel's Hadoop benchmark tools [50] and data sampled from the Common Crawl [34] document corpus. The set of aggregation queries and the UDFs in the benchmark are characteristic of the workload used at Conviva.

For each query in our dataset, we compute the true values of $\theta(D)$ and the true confidence interval and then run the query on 100 different samples of fixed size $n = 1,000,000$[7] rows. For each run, we compute a confidence interval using an error estimation technique, from which we get δ. We then pick a reasonable range of $[-0.2, 0.2]$, and if δ is outside this range for at least 5% of the samples, we declare error estimation a failure for the query. We present separately the cases where $\delta > 0.2$ (i.e., *pessimistic* error estimation) and where $\delta < -0.2$ (i.e., *optimistic* and *incorrect* error estimation). In addition, since only queries with `COUNT, SUM, AVG,` and `VARIANCE` aggregates are amenable to closed-form error estimation, while all aggregates are amenable to the bootstrap, 43.21% of Facebook queries and 62.79% Conviva queries can only be approximated using bootstrap based error estimation methods. Results are displayed in Fig. 4.2.

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

4.5 Conclusion

It is clear from this evaluation that no type of error estimation gives completely satisfactory results. Lacking a one-size-fits-all error estimation technique, we need an algorithm to identify the cases where the non-conservative methods work and the cases where we must use conservative large deviation bounds or avoid sampling altogether. We call such a procedure a *diagnostic*. These diagnostic procedures will be the focus of the next chapter.

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

Chapter 5

Error Diagnostics

In this chapter, we are going to use a diagnostic recently developed by Kleiner et al. [52], but first let us provide some intuition for how a diagnostic should work. We consider first an impractical *ideal* diagnostic. To check whether a particular kind of error estimation for a query θ on data S is likely to fail, we could simply perform the evaluation procedure we used to present results in the previous section. That is, we could sample repeatedly (say $p = 100$ times) from the underlying dataset D to compute the true confidence interval, estimate a confidence interval on each sample using the error estimation method of interest, compute δ for each one, and check whether most of the δs are close to 0. If we use sufficiently many samples, this ideal procedure will tell us exactly what we want to know. However, it requires repeatedly sampling large datasets from D (and potentially executing queries on K bootstrap resamples for each of these samples), which is prohibitively expensive.

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

Instead, we can also approximate the ideal diagnostic by performing it on a sequence of much smaller samples and extrapolating the results to the actual sample, S. This is the basis of the diagnostic procedure of Kleiner et al. [52]. It is motivated computationally by the following observation: If S is a simple random sample from D, then subsamples generated by disjointly partitioning S are themselves mutually independent simple random samples from D. Thus by partitioning S we can identify small samples from D without performing

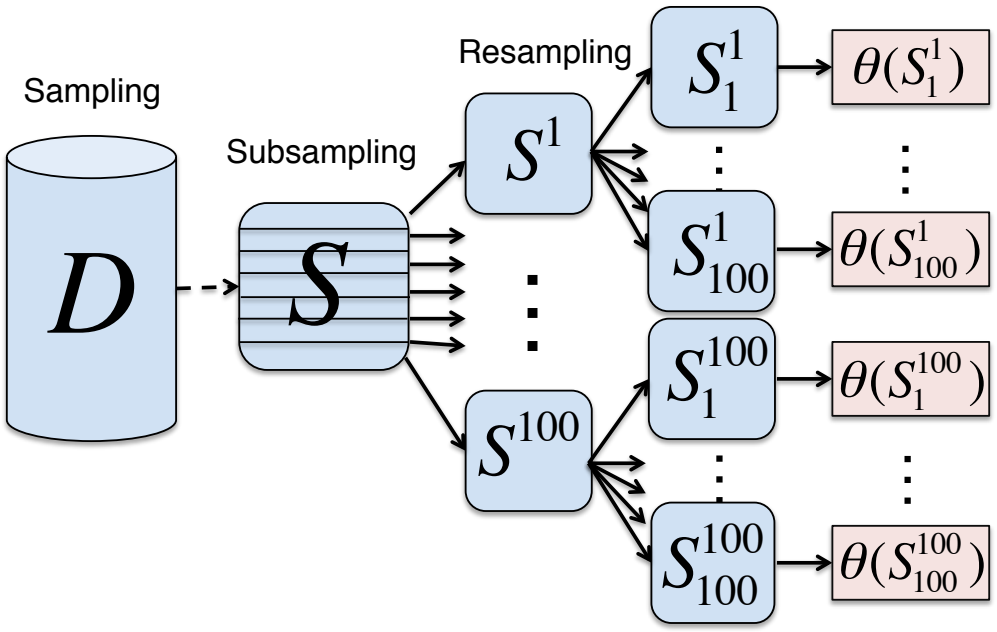


Figure 5.1: The pattern of computation performed by the diagnostic algorithm for a single subsample size. Here the procedure being checked is the bootstrap, so each subsample is resampled many times. For each subsample S^j, the bootstrap distribution $(\theta(S_1^j), \theta(S_2^j), ...)$ is used to compute an estimate of ξ, which is assessed for its closeness to the value of ξ computed on the distribution of values $(\theta(S^1), \theta(S^2), ...)$. The computation pictured here is performed once for each subsample size b_i, using the same sample S.

additional I/O. Of course, the effectiveness of an error estimation technique on small samples may not be predictive of its effectiveness on S. Therefore careful extrapolation is necessary: we must perform the procedure at a sequence of increasing sample sizes, $(b_1, ..., b_k)$ and check whether the typical value of δ decreases with b_i and is sufficiently small for the largest sample size b_k. The disjointness of the partitions imposes the requirement that each sample size b_i, multiplied by the number of samples p, be smaller than S.

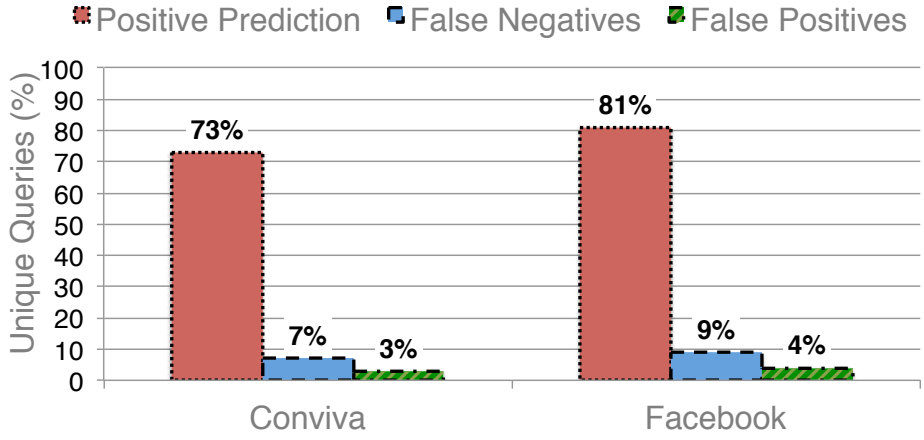
5.1 Kleiner et al.'s Diagnostics

The details of the checks performed by the diagnostic are not critical for the remainder of this chapter. The diagnostic algorithm, as applied to query approximation, is described precisely in Algorithm 1. We provide it here for completeness. The algorithm requires a large number of parameters. In our experiments, we have used settings similar to those suggested by Kleiner et al.: $p = 100$, $k = 3$, $c_1 = 0.2$, $c_2 = 0.2$, $c_3 = 0.5$, and $\beta = 0.95$ on subsamples whose rows have total size 50MB, 100MB and 200MB. Fig. 5.1 depicts the pattern of computation performed by the diagnostic.

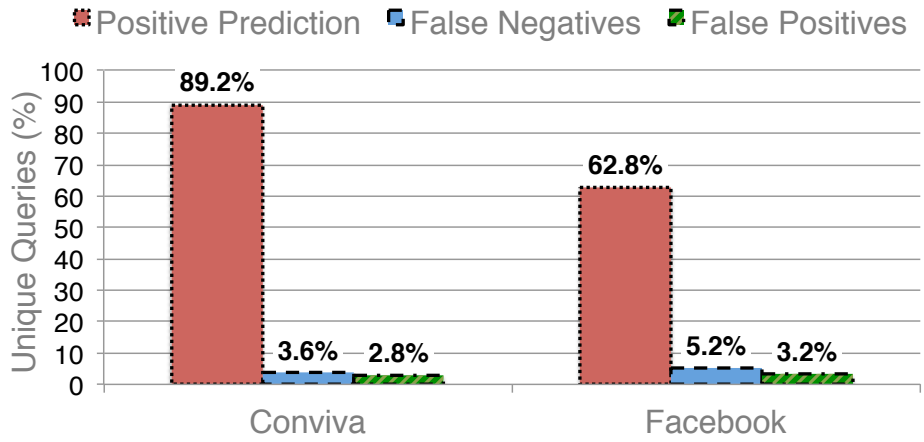
Input: $S = (S_1, ..., S_n)$: a sample of size n from D
θ: the query function
α: the desired coverage level for confidence intervals
ξ: a function that produces confidence interval estimates given a sample, a query function, and a coverage level, e.g. the bootstrap
b_1, ..., b_k: an increasing sequence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

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

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



(a) Closed Form Diagnostic Experiments



(b) Bootstrap Diagnostic Experiments

Figure 5.2: Figures 5.2a and 5.2b compare the diagnostic prediction accuracy for Closed Form and Bootstrap error estimation respectively. For 5.2a, we used a workload of 100 queries each from Conviva and Facebook that only computed `AVG, COUNT, SUM` or `VARIANCE` based aggregates. For 5.2b we used a workload of 250 queries each from Conviva and Facebook that computed a variety of complex aggregates instead.

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

5.2 Diagnosis Accuracy

We demonstrate the diagnostic's utility in query approximation by providing a thorough evaluation of its accuracy on Facebook and Conviva queries for both bootstrap and closed form error estimation. We used two different datasets to evaluate the diagnostic's effectiveness on real world workloads. The first dataset was derived from a subset of 350 Apache Hive [74] queries from Conviva Inc. [36]. 100 of these queries computed `AVG, COUNT, SUM` or `VARIANCE`, and the remaining 250 included more complicated expressions for which error estimates could be obtained via the bootstrap. This dataset was 1.7 TB in size and consisted of 0.5 billion records of media accesses by Conviva users. The second dataset was one derived from Facebook Inc. and was approximately 97.3 TB in size, spanning over 40 billion records in different tables. As with the Conviva queries, we picked sets of 100 and 250 queries for evaluation respectively suited for closed-forms and bootstrap based error estimation techniques. The performance of the diagnostic algorithm is a function of a variety of statistical parameters all of which affect the accuracy of the underlying diagnosis. Fig. 5.2 compares the success accuracy of the diagnostics over the set of queries from Facebook and Conviva for our experiments. Overall, with these diagnostic settings in place, we predicted that 84.57% of Conviva queries and 68% of Facebook queries resulted in a *positive prediction* while the others cannot, with less than 3.1% false positives and 5.4% false negatives.

5.3 Conclusion

The past decade has seen several investigations of *diagnostic* methods in the statistics literature [26, 52]. In this chapter, we extend the diagnostic algorithm proposed by Kleiner et al. [52] to validate multiple procedures for generating error bars at runtime. However, this algorithm was not designed with computational considerations in mind and involves tens of thousands of test query executions, making it prohibitively slow in practice. In the next chapter, we present a series of optimization techniques across all stages of the query processing pipeline that reduce the running time of the diagnostic algorithm from hundreds of seconds to only a couple of seconds, making it a practical tool in a distributed AQP system. As a demonstration of the utility of the diagnostic, we integrate bootstrap-based error estimation techniques and the diagnostics into BlinkDB. With the diagnostic in place, we demonstrate that BlinkDB can answer a range of *complex* analytic queries on large samples of data (of gigabytes in size) at *interactive speeds* (i.e., within a couple of seconds), while falling back to non-approximate methods to answer queries whose errors cannot be accurately estimated.

Chapter 6

An Architecture for Approximate Query Execution

Finally, at this point, we have all the necessary ingredients to sketch the entire query approximation pipeline. Fig. 6.1 describes an end-to-end workflow of a large-scale distributed approximate query processing framework that computes approximate answers, estimates errors and verifies its correctness. The *Query (I)* is first compiled into a *Logical Query Plan (II)*. This in turn has three distinct parts: one that computes the approximate answer "$\theta(S)$", another that computes the error "$\hat{\xi}$" and finally the component that computes the diagnostic tests. Each logical operator is instantiated by the *Physical Query Plan (III)* as a DAG (Directed Acyclic Graph) of tasks. These tasks execute and communicate in parallel, and operate on a set of samples that are distributed across multiple disks or cached in the memory. Finally, the *Data Storage Layer (IV)* is responsible for efficiently distributing these samples across machines and deciding which of these samples to cache in memory. Notice that there are several steps that are necessary in computing queries on sampled data. If we are to fulfill our promise of interactive queries, each of these steps must be fast. In particular, hundreds of bootstrap queries and tens of thousands of small diagnostic queries must be performed within seconds. In this chapter, we will first describe the *Poissonized Resampling* technique (§6.1) that enables us to create multiple resamples efficiently by simply streaming the tuples of the original sample. Then, we present a baseline solution based on this technique (§6.2), and finally propose a number of query plan optimizations (§6.3) to achieve interactivity.

6.1 Poissonized Resampling

Recall that the bootstrap (including the bootstraps performed on the small subsamples used in the diagnostic) requires the identification of many *resampled* datasets from a given sample S. To compute a resample from S, we take $|S|$ rows with replacement from it. Equivalently, we can also view resampling as assigning a random count in $\{0, 1, ..., |S|\}$ to

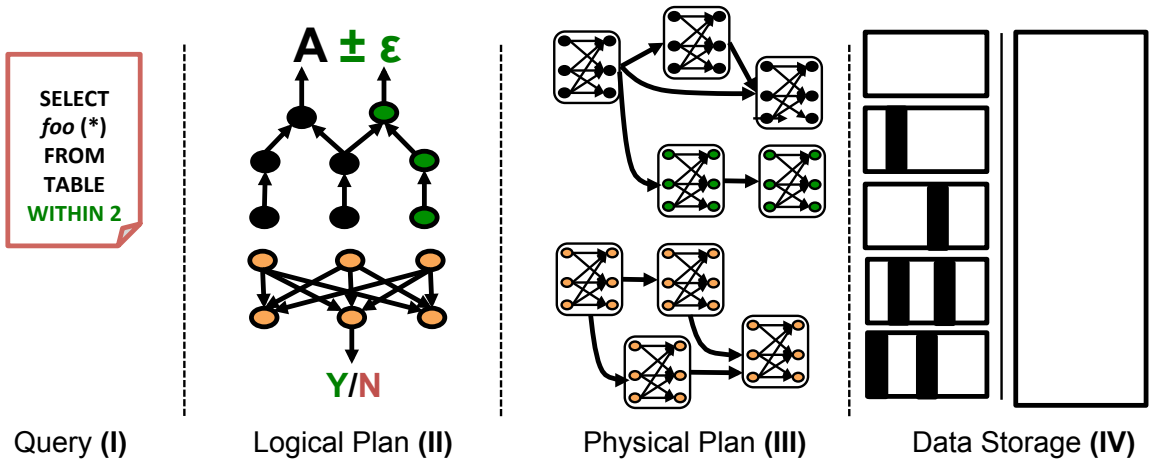


Figure 6.1: Workflow of a Large-Scale, Distributed Approximate Query Processing Framework.

each row of S according to a certain distribution, with the added constraint that the sum of counts assigned to all rows is exactly $|S|$. The distribution of counts for each row is `Poisson(1)`; this distribution has mean 1, and counts can be sampled from it quickly [64]. However, the sum constraint of having exactly $|S|$ tuples couples the distribution of row counts and substantially complicates the computation of resamples, especially when samples are partitioned on multiple machines. The coupled resample counts must be computed from a large multinomial distribution and stored, using $O(|S|)$ memory per resample. Pol and Jermaine showed that a Tuple Augmentation (TA) algorithm designed to sample under this constraint not only incurred substantial pre-processing overheads but was also in general $8 - 9\times$ slower than the non-bootstrapped query [66].

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

[1]Please see Chapter 3.7 of [78] for a discussion of Poissonization and the bootstrap.

6.2 Baseline Solution

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

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

Implementing error estimation or the diagnostics (for both bootstrap and closed forms) in the query layer similarly involves either plugging in appropriate error estimation formulas or writing a query to execute subqueries on small samples of data and compare the estimated error with the true error, respectively. With this scheme, the bootstrap requires execution of 100 separate subqueries, and a diagnostic query requires $30,000$ subqueries (we use $K = 100$ bootstrap resamples, and set $p = 100$ and $k = 3$ in execution of Algorithm 1 settings). Unfortunately, the overhead introduced by executing such a large number of subqueries compromise the interactivity of this approach, even when leveraging distributed in-memory data processing frameworks such as Shark [81] and Spark [84].

There are several overheads incurred by this naïve solution. First, both bootstrap and diagnostics are performing the same set of queries over and over again on multiple samples of data. Second, each query further gets compiled into one or more tasks. As the number

of tasks grows into thousands, the per-task overhead and the contention caused by each task in continuously resampling the same sample adds up substantially. This suggests that achieving interactivity may require changes to the entire query processing framework.

Prior work, including TA/ODM [66] and EARL [53] have proposed several optimizations to reduce the repetitive work of the bootstrap. While we build on some of these techniques, none of them aim to provide interactive response times. For example, EARL is built to run on top of Hadoop MapReduce, which (at least at the time of that work) was unable to run even small queries interactively due to scheduling overhead and I/O costs. On the other hand, *Tuple Augmentation* (TA) and *On-Demand Materialization* (ODM) based algorithms incur substantial overheads to create exact samples with replacement. Next, we will show that leveraging Poissonized resampling techniques to create resamples from the underlying data not only alleviates the need for directly building on these solutions but it is also orders of magnitude more efficient both in terms of runtime and resource usage.

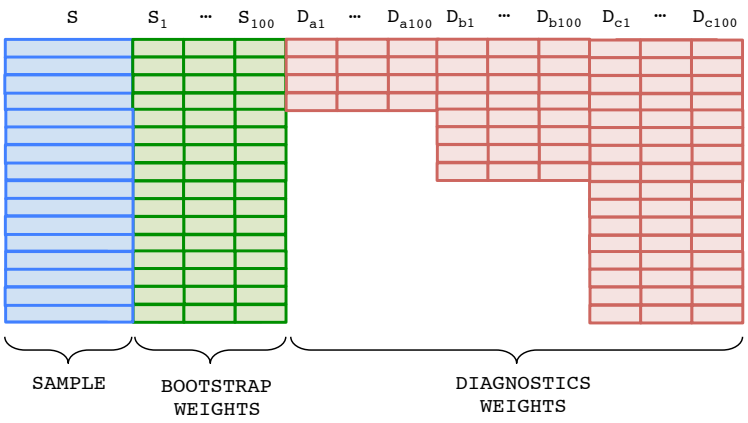
6.3 Query Plan Optimizations

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

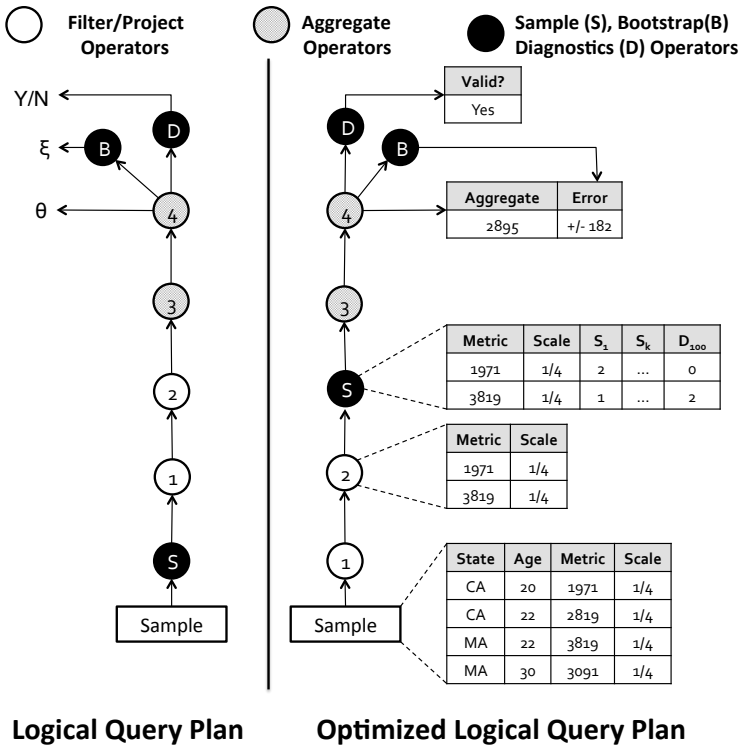
Scan Consolidation

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

To mitigate these problems, we reduce this problem to one that requires a *single scan* of the original sample to execute all bootstrap sub-queries to estimate the error and all diagnosis sub-queries to verify the error accuracy. To achieve this goal, we first optimize the logical plan by extending the simple Poissonized Resampling operator in §6.2 to augment the tuple simultaneously with multiple sets of resampling weights – each corresponding to a resample that may either be required to estimate the error using bootstrap or to verify the accuracy of estimation using the diagnostics. More specifically, as shown in Fig. 6.2a, to



(a) Scan Consolidation



(b) Rewriting the Logical Query Plan

Figure 6.2: Logical Query Plan Optimizations.

estimate sampling error using bootstrap, we augment each tuple in sample S by associating a set of 100 independent weights S_1, ..., S_{100}, each drawn from a `Poisson(1)` distribution to create 100 resamples of S. For the diagnostics, we first logically partition[2] the sample S

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

into multiple sets of 50 MB, 100 MB and 200 MB and then associate the weights D_{a1}, ..., D_{a100}; D_{b1}, ..., D_{b100} and D_{c1}, ..., D_{c100} to each row in order to create 100 resamples for each of these three sets. Overall, we use 100 instances of these three sets for accurate diagnosis. Executing the error estimation and diagnostic queries in a *single pass* further enables us to leverage a lot of past work on efficiently scheduling multiple queries by sharing a single cursor to scan similar sets of data [9, 44, 80].

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

Operator Pushdown

Ideally, as shown in Fig. 6.2b (left), the *Poissonized resampling operator* should be inserted immediately after the `TABLESCAN` operator in the query graph and the *bootstrap* and *diagnostic* operators should be inserted after the final set of aggregates. However, this results in wasting resources on unnecessarily maintaining weights of tuples that may be filtered upstream before reaching the aggregates. Therefore, to further optimize the query plan, we added a *logical plan rewriter* that rewrites the logical query plan during the optimization phase. Rewriting the query plan involves two steps. First, we find the longest set of consecutive *pass-through*[3] operators in the query graph. Second, we insert the custom *Poissonized resampling operator* right before the first non *pass-through* operator in the query graph. This procedure is illustrated in Fig. 6.2b, wherein we insert the *Poissonized resampling operator* between stage 2 and 3. The subsequent aggregate operator(s) is(are) modified to compute a set of resample aggregates by appropriately scaling the coresponding aggregation column with the weights associated with every tuple. These set of resample aggregates are then streamed into the *bootstrap* and *diagnostic* operators.

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

sample.

[3]We loosely define *pass-through* as those set of operators that do not change the statistical properties of the set of columns that are being finally aggregated. These operators are relatively simple to identify during the query planning and analysis phase [21] and consist of, but are not limited to `scans, filters, projections,` etc.

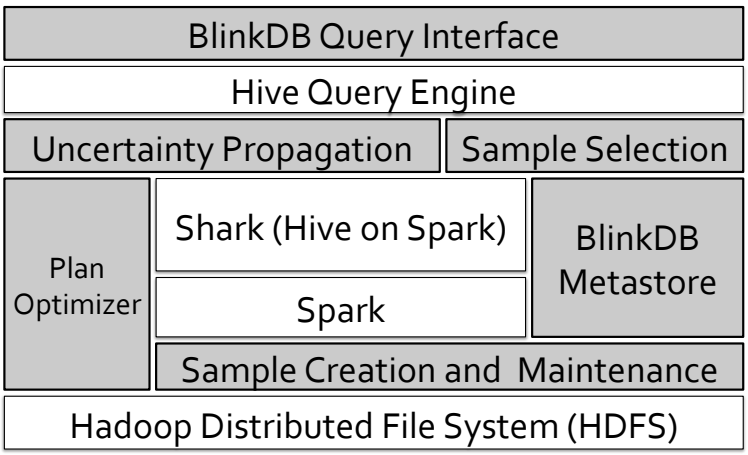


Figure 6.3: BlinkDB System Architecture.

results in overall error estimation overhead being reduced by several orders of magnitude using the bootstrap.

6.4 Performance Tradeoffs

Once again, we implemented our solution in BlinkDB which is built on top of Shark [81], a query processing framework, which in turn is built on top of Spark [84], an execution engine. Recall that Shark supports caching inputs and intermediate data in fault tolerant data structures called RDDs (Resilient Distributed Datasets). BlinkDB leverages Spark and Shark to effectively cache samples in memory and to allow users to pose SQL-like aggregation queries (with response time or error constraints) over the underlying data, respectively. Queries over multiple terabytes of data can hence be answered in seconds, accompanied by meaningful error bars bracketing the answer that would be obtained if the query ran instead on the full data. The basic approach taken by BlinkDB is exactly the same as described above—it precomputes and maintains a carefully chosen collection of samples of input data, selects the best sample(s) at runtime for answering each query, and provides meaningful error bounds using statistical sampling theory. Furthermore, we have also helped in successfully integrating the same set of techniques in Facebook's Presto [42], another well-known open-source distributed SQL query engine.

As described above, we first added the *Poissonized Sampling* operator, the *bootstrap* operator, and the *diagnostics* operator in BlinkDB, then modified the aggregate functions to work on weighted tuples. Finally, we implemented the logical plan rewriter to optimize the query plan for error estimation and diagnostics. While the aforementioned query plan optimizations brought the end-to-end query latency to tens of seconds, to achieve interactivity,

we need a fine grain control on 3 key aspects of the physical plan– the query's degree of parallelism, per-task data locality, and straggler mitigation.

Degree of Parallelism

BlinkDB maintains a variety of disjoint random samples of the underlying data that is cached across a wide range of machines. These samples are randomly shuffled across the cluster and any subset of a particular sample is a random sample as well. Given that the ability to execute the query on any random sample of data (on any subset of machines) makes the implementation of both the error estimation and diagnostic procedures embarrassingly parallel (except the aggregation step at the end), parallelizing the logical query plan to run on many machines (operating on a fraction of data) significantly speeds up end-to-end response times. However, given the final aggregation step, arbitrarily increasing the parallelism often introduces an extra overhead, even in an interactive query processing framework like BlinkDB. A large number of tasks implies additional per-task overhead costs, increased many-to-one communication overhead during the final aggregation phase, and a higher probability of stragglers. Striking the right balance between the degree of parallelism and the overall system throughput is primarily a property of the underlying cluster configuration and the query workload. We will revisit this trade-off in §6.5.

Per-Task Data Locality

While executing a query on a sample, we ensure that a large portion of the samples are cached in cluster memory, using BlinkDB's caching facilities. However we observed empirically that using all the available RAM for caching inputs is not a good idea. In general, given that the total RAM in a cluster is limited, there is a tradeoff between caching a fraction of input data versus caching intermediate data during the query's execution. Caching the input invariably results in faster scan times, but it decreases the amount of per-slot memory that is available to the query during execution. While this is once again a property of the underlying cluster configuration and the query workload, we observed that caching a fixed fraction of samples and allotting a bigger portion of memory for caching intermediate data during the query execution results in the best average query response times. We will revisit this trade-off in §6.5.

Straggler Mitigation

With a fine-grained control over the degree of parallelism and per-task locality, avoiding straggling tasks during query execution results in further improvements in query runtime. In order to reduce the probability of a handful of straggling tasks slowing down the entire query, we always spawn 10% more tasks on identical random samples of underlying data on a different set of machines and, as a result, do not wait for the last 10% tasks to finish. Straggler mitigation does not always result in end-to-end speedups and may even, in theory, introduce

bias in error estimation when long tasks are systematically different than short ones [65]. However, in our experiments we observed that straggler mitigation speeds up queries by hundreds of milliseconds and causes no deterioration in the quality of our results.

6.5 Evaluation

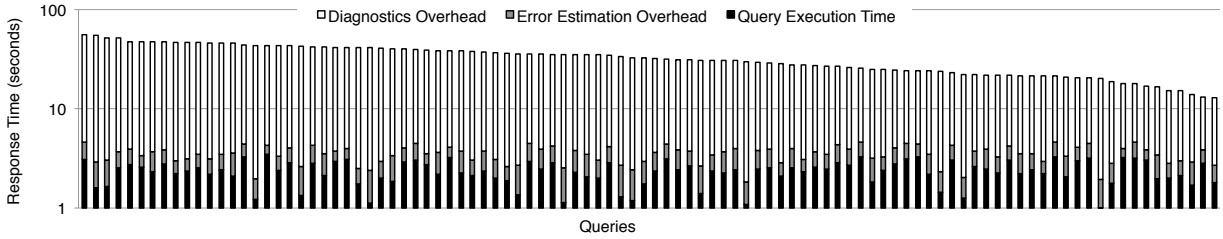
We evaluated our performance on two different sets of 100 real-world Apache Hive queries from the production clusters at Conviva Inc. [36]. The queries accessed a dataset of size 17 TB stored across 100 Amazon EC2 *m1.large* instances (each with 4 ECUs[4] (EC2 Compute Units), 7.5 GB of RAM, and 840 GB of disk). The cluster was configured to utilize 75 TB of distributed disk storage and 600 GB of distributed RAM cache. The two query sets consists of:

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

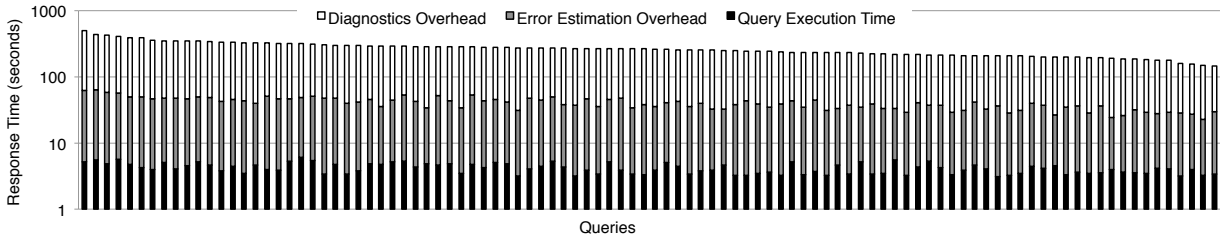
Baseline Results

Fig. 6.4a and Fig. 6.4b plot the end-to-end response times for executing the query on a sample, estimating the error and running the diagnostics using the naive implementation described in §6.2 on `QSet-1` and `QSet-2`, respectively. Every query in these `QSet`s is executed with a 10% error bound on a cached random sample of at most 20 GB in size from the underlying 17 TB of data. Given that each of the 3 steps—the query execution on the sample, the error estimation and running the diagnosis—happens in parallel, for each query we plot the *Query Response Time* (i.e., the time it takes to execute the query on a sample), the *Error Estimation Overhead* (i.e., the *additional* overhead of estimating bootstrap or closed form error), and the *Diagnostics Overhead* (i.e., the *additional* overhead of running the diagnosis) separately. These results clearly highlight that simply rewriting the queries to implement error estimation and diagnostics on even relatively small sample sizes typically takes several minutes to run (and more importantly costs 100× to 1000× more resources), making it too slow for interactivity and hugely inefficient. Next, we will revisit our optimizations from Sections §6.3 and §6.4, and demonstrating their individual speedups in response times with respect to our baseline solution.

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



(a) `QSet-1`



(b) `QSet-2`

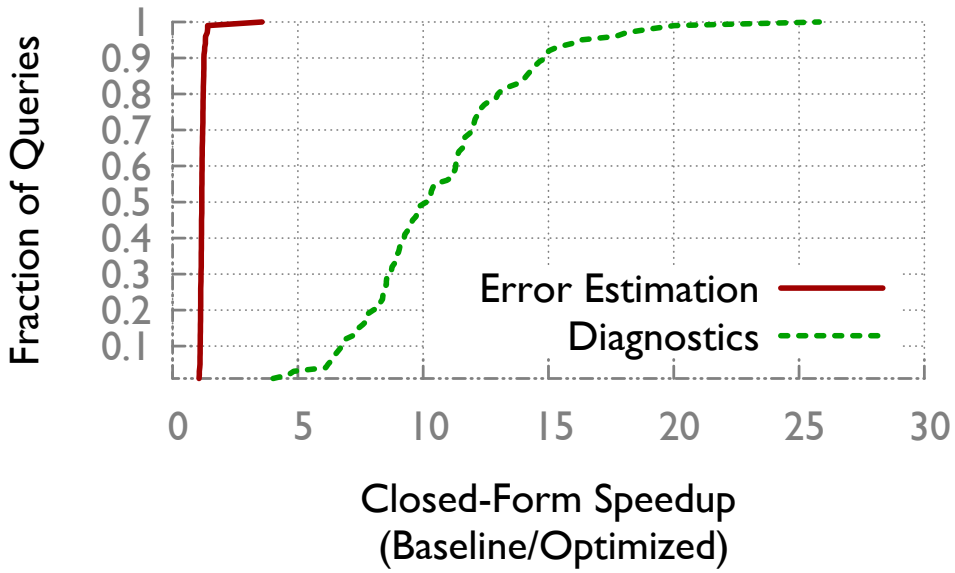
Figure 6.4: 6.4a and 6.4b show the naïve end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for `QSet-1` (i.e., set of 100 queries which can be approximated using closed-forms) and `QSet-2` (i.e., set of 100 queries that can only be approximated using bootstrap), respectively. Each set of bars represents a single query execution with a 10% error bound.

Query Plan Optimizations

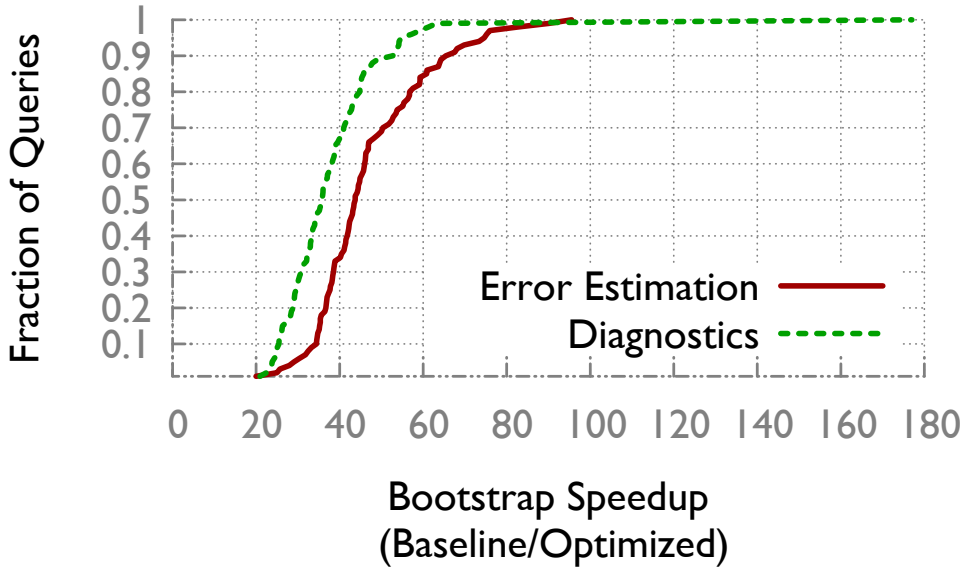
As expected, some of our biggest reductions in end-to-end query response times come from *Scan Consolidation* and *Operator Pushdown*. Fig. 6.5a and Fig. 6.5b show the cumulative distribution function of speedups yielded by query plan optimization techniques (*Scan Consolidation* and *Operator Pushdown*) for error estimation and diagnostics (larger overhead ratios indicate greater speedups). Individually, Fig. 6.5a and Fig. 6.5b show the speedups associated with `QSet-1` & `QSet-2` respectively. These speedups are calculated with respect to the baseline established in Fig. 6.4a and Fig. 6.4b and demonstrate improvements of $1-2\times$ and $5-20\times$ (for error estimation and diagnostics respectively) in `QSet-1`, and $20-60\times$ and $20-100\times$ (for error estimation and diagnostics respectively) in `QSet-2`.

Performance Tuning

With the query plan optimizations in place, we will next show the benefits of tuning the underlying physical execution plan by varying the degree of parallelism and input cache sizes. As we explain in §6.4, we observed that striking the right balance between the degree of parallelism and the overall system throughput is primarily a property of the underlying cluster configuration and the query workload. We verified this observation empirically by



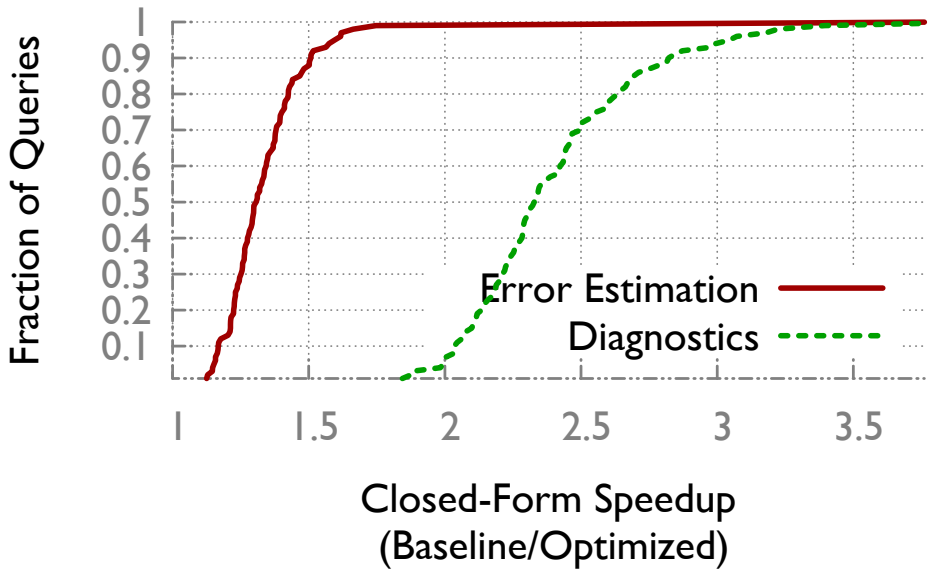
(a) `QSet-1`



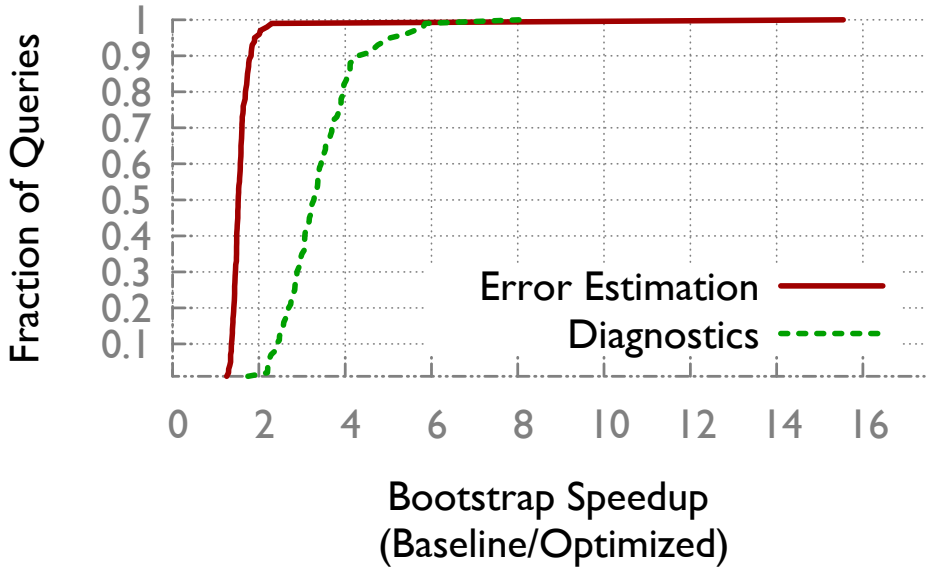
(b) `QSet-2`

Figure 6.5: Fig. 6.5a and Fig. 6.5b show the cumulative distribution function of speedups yielded by query plan optimizations (i.e., *Scan Consolidation* and *Sampling Operator Push-down*) for error estimation and diagnostics with respect to the baseline defined in §6.2.

varying the amount of maximum parallelism for each query. As shown in Fig. 6.7a, in our experiments, we observed that both bootstrap-based error estimation and the diagnostic

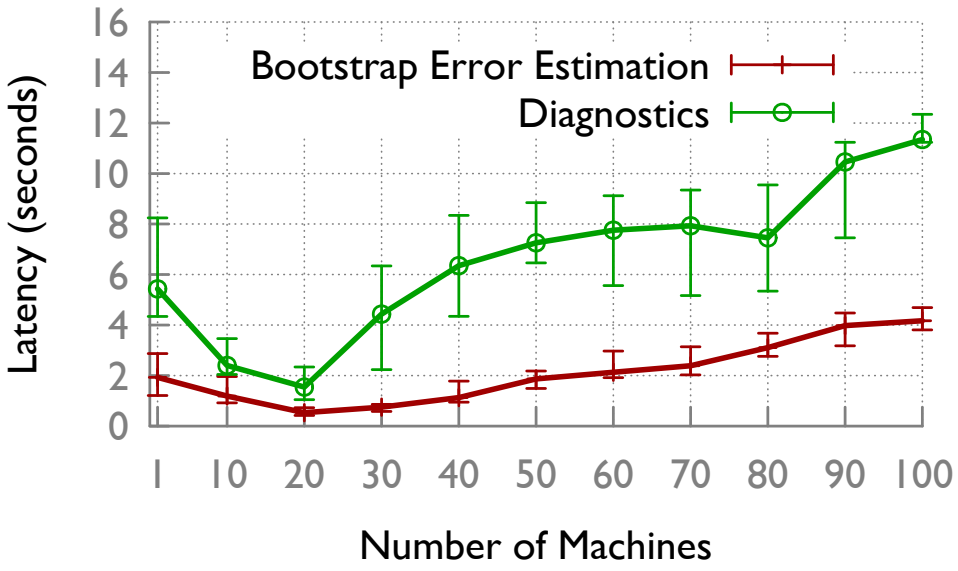


(a) `QSet-1`

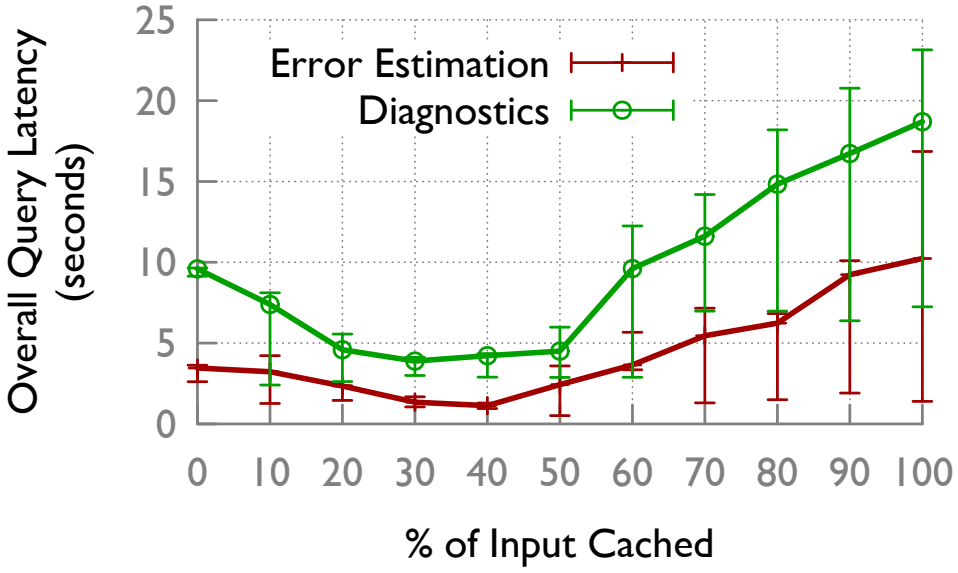


(b) `QSet-2`

Figure 6.6: Fig. 6.6a and Fig. 6.6b show the speedups yielded by a fine grained control over the physical plan (i.e., bounding the query's *degree of parallelism*, *size of input caches*, and *mitigating stragglers*) for error estimation and diagnostics with respect to the baseline defined in §6.3.



(a) `QSet-1 + QSet-2`



(b) `QSet-1 + QSet-2`

Figure 6.7: Fig. 6.7a and Fig. 6.7b demonstrate the trade-offs between the bootstrap-based error estimation/diagnostic techniques and the number of machines or size of the input cache, respectively (averaged over all the queries in `QSet-1` and `QSet-2` with vertical bars on each point denoting 0.01 and 0.99 quantiles).

procedure were most efficient when executed on up to 20 machines. Increasing the degree of parallelism may actually lead to worse performance as the task scheduling and communication overheads offsets the parallelism gains. Again, note that the optimal degree of parallelism we report here is an artifact of our query load and the sample sizes we picked for our diagnostic procedure. Choosing the degree of parallelism automatically is a topic of future work.

Similarly, we observed that caching the entire set of input samples is not always optimal. As we explain in §6.4, given that the total RAM in a cluster is limited, there is a tradeoff between caching a fraction of input data versus intermediate data during query execution. In our experiments we observed that caching a fixed fraction of samples and allotting a bigger portion of memory for query execution results in the best average query response times. Fig. 6.7b shows the tradeoff graph with the percentage of samples cached on the x-axis and the query latencies on the y-axis. In our case, we achieve the best end-to-end response times when $30 - 40\%$ of the total inputs were cached, accounting for roughly $180 - 240$ GB of aggregate RAM.

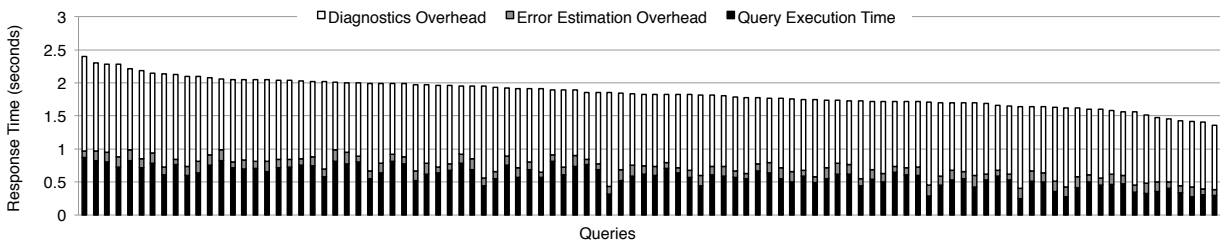
Fig. 6.6a and Fig. 6.6b show the cumulative distribution function of the speedup for error estimation and diagnostics obtained by (i) tuning the degree of parallelism, (ii) the fraction of samples being cached (as discussed above), and, in addition, (iii) increasing the number of sub-queries by 10% to account for straggler mitigation. Specifically, 6.6a and 6.6b show the speedups associated with `QSet-1` (i.e., set of 100 queries that can be approximated using closed-forms) and `QSet-2` (i.e., set of 100 queries that can only be approximated using bootstrap), respectively. Note that the baseline for these experiments is the implementation of BlinkDB with the query plan optimizations described in §6.3, and not the naive implementation.

Putting it all Together

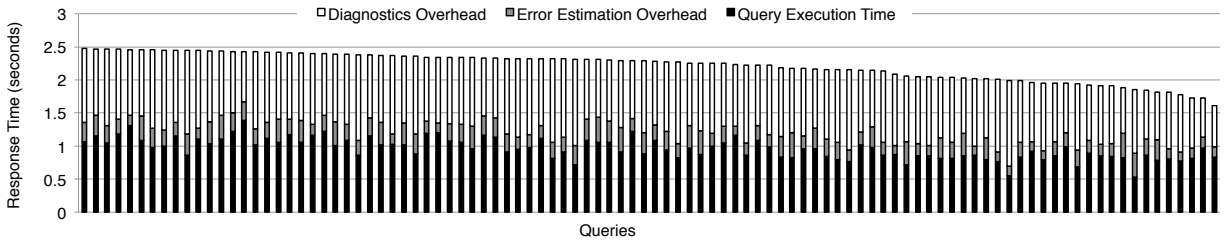
With all the optimizations in place, Fig. 6.8a and Fig. 6.8b show the per-query overheads of error estimation and diagnostics on our two query sets `QSet-1` and `QSet-2`, respectively. Note that in comparison to Fig. 6.4a and Fig. 6.4b, we have been able to improve the execution times by $10 - 200\times$, and achieve end-to-end response times of a few seconds, thus effectively providing interactivity.

6.6 Conclusion

In this chapter, we presented several optimizations that make the diagnostic and the procedures that generate error bars practical, ensuring that these procedures do not affect the interactivity of the overall query. With these optimizations, and leveraging recent systems for low-latency exact query processing, we demonstrated a viable end-to-end system



(a) `QSet-1`



(b) `QSet-2`

Figure 6.8: Fig. 6.8a and Fig. 6.8b show the optimized end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for `QSet-1` (i.e., set of 100 queries which can be approximated using closed-forms) and `QSet-2` (i.e., set of 100 queries that can only be approximated using bootstrap), respectively. Each set of bars represents a single query execution with a 10% error bound.

for approximate query processing using sampling. We showed that this system can deliver interactive-speed results for a wide variety of analytic queries from real world production clusters.

6.7 Appendix

In the interest of reproducibility, this section details all per-query response times, error estimation overheads and the diagnostic overheads for queries in `QSet-1` and `QSet-2`. Please note that while we were unable to disclose the exact queries and datasets used in these experiments due to the proprietary nature of these workloads, given that most of these queries were I/O bounded, these overheads only depend on the size of the underlying sample (i.e., 20GB) and *not* on the exact nature of these workloads.

Naive Approach

Tables 6.1 and 6.2 show the naïve end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for `QSet-1` (i.e., set of 100

queries which can be approximated using closed-forms) and `QSet-2` (i.e., set of 100 queries that can only be approximated using bootstrap), respectively.

Table 6.1: The naïve end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for (`QSet-1`). Each row represents a single query execution with a 10% error bound.

#	Query Execution (s)	Error Estimation (s)	Diagnostics (s)	Total (s)
1	5.15	57.40	441.71	504.26
2	5.61	58.20	372.90	436.72
3	4.86	54.25	372.54	431.65
4	5.75	51.64	355.93	413.33
5	4.72	45.53	347.71	397.96
6	4.22	45.59	343.06	392.87
7	3.94	42.79	314.76	361.49
8	5.07	43.32	304.12	352.52
9	4.08	44.03	304.11	352.23
10	4.52	42.78	301.86	349.15
11	5.25	44.73	299.15	349.13
12	4.67	44.70	295.34	344.70
13	3.79	39.23	296.48	339.51
14	4.45	40.89	290.24	335.58
15	3.46	40.48	288.36	332.31
16	4.61	34.98	292.03	331.61
17	3.99	46.66	279.79	330.44
18	3.90	42.95	277.92	324.77
19	5.32	41.61	276.93	323.86
20	6.07	42.60	274.91	323.58
21	5.50	45.13	266.12	316.75
22	3.42	44.57	262.44	310.43
23	4.71	42.95	254.40	302.06
24	3.42	37.01	261.41	301.85
25	3.83	37.77	257.37	298.97
26	4.82	40.66	250.90	296.38
27	4.79	31.37	258.95	295.10
28	5.23	39.66	249.67	294.56
29	5.35	48.72	238.73	292.80
30	4.36	38.35	248.17	290.87
31	4.90	29.60	255.04	289.55
32	4.61	47.41	236.75	288.78
33	4.88	38.85	243.38	287.11

34	3.49	30.51	252.48	286.48
35	4.76	48.20	232.15	285.11
36	4.23	39.41	238.84	282.49
37	5.04	40.22	235.50	280.76
38	4.86	37.17	237.65	279.68
39	3.16	27.93	247.14	278.23
40	4.05	44.19	228.58	276.82
41	4.47	39.82	231.04	275.33
42	5.27	45.25	223.46	273.98
43	4.35	34.08	234.14	272.57
44	3.18	34.12	234.70	272.00
45	3.85	42.52	224.49	270.86
46	3.42	32.51	234.58	270.51
47	5.24	40.95	223.38	269.57
48	3.89	43.46	220.60	267.94
49	3.37	30.75	233.64	267.76
50	3.32	35.25	227.89	266.47
51	3.86	31.75	227.63	263.25
52	5.12	35.54	219.55	260.22
53	4.44	38.61	217.06	260.11
54	3.40	31.98	224.71	260.08
55	3.84	36.07	219.99	259.90
56	3.91	28.59	222.53	255.02
57	4.64	28.30	218.44	251.38
58	3.28	34.60	212.49	250.37
59	3.26	40.62	204.75	248.62
60	3.44	35.30	209.67	248.41
61	3.63	31.24	213.00	247.88
62	3.26	35.52	202.90	241.68
63	5.20	38.45	193.86	237.51
64	3.34	31.91	201.21	236.45
65	3.69	41.25	190.90	235.84
66	3.24	28.34	203.23	234.82
67	4.64	28.60	201.47	234.71
68	3.43	34.08	195.22	232.73
69	5.23	29.55	197.42	232.21
70	3.43	35.29	188.53	227.25
71	3.44	30.06	189.50	223.00
72	5.55	27.99	188.78	222.32

73	3.22	25.98	191.88	221.08
74	4.34	36.15	179.05	219.53
75	5.36	31.94	177.80	215.09
76	4.21	32.78	176.98	213.97
77	3.32	25.68	184.47	213.47
78	3.91	27.02	181.17	212.11
79	4.66	37.08	169.68	211.42
80	4.06	28.45	178.68	211.20
81	3.12	33.86	174.06	211.04
82	3.24	25.57	179.40	208.22
83	3.46	27.59	176.98	208.03
84	4.43	36.01	163.92	204.37
85	4.18	33.36	165.01	202.54
86	4.54	22.17	175.64	202.36
87	3.29	31.91	165.89	201.09
88	3.64	32.98	162.78	199.40
89	3.47	25.41	168.60	197.47
90	3.55	32.78	159.77	196.10
91	3.98	20.48	166.61	191.07
92	3.66	22.47	163.41	189.54
93	3.53	28.73	155.92	188.17
94	3.44	25.85	155.73	185.02
95	4.16	23.78	152.83	180.77
96	4.09	25.27	149.32	178.68
97	3.17	25.10	132.87	161.14
98	3.94	23.16	129.23	156.33
99	3.28	19.36	127.45	150.09
100	3.39	26.35	117.45	147.20

Table 6.2: The naïve end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for (`QSet-2`). Each row represents a single query execution with a 10% error bound.

#	**Query Execution (s)**	**Error Estimation (s)**	**Diagnostics (s)**	**Total (s)**
1	3.11	1.50	51.27	55.88
2	1.60	1.32	52.53	55.45
3	1.65	1.41	49.03	52.09
4	2.53	1.14	48.21	51.88
5	2.75	1.18	44.07	48.01
6	2.60	0.79	44.47	47.87
7	2.33	1.35	44.15	47.83
8	2.77	1.11	43.78	47.66
9	2.21	0.81	43.84	46.86
10	2.35	0.79	43.51	46.65
11	2.56	0.92	43.15	46.63
12	2.20	0.92	43.32	46.44
13	2.45	1.05	42.83	46.33
14	2.10	1.52	42.68	46.30
15	3.27	1.17	39.85	44.30
16	1.23	0.75	41.93	43.90
17	3.48	0.79	39.53	43.80
18	2.39	0.93	40.23	43.55
19	2.87	1.20	39.37	43.43
20	1.35	1.28	40.03	42.65
21	2.82	1.51	37.82	42.14
22	2.13	1.43	38.53	42.09
23	2.97	0.77	38.10	41.85
24	3.11	0.87	37.52	41.50
25	1.76	0.75	38.91	41.43
26	1.13	1.28	39.01	41.42
27	2.01	0.97	38.18	41.15
28	1.87	1.52	36.92	40.31
29	2.91	1.16	36.19	40.25
30	3.05	1.43	35.50	39.98
31	2.74	0.78	35.87	39.39
32	2.20	1.45	35.10	38.75
33	3.26	0.83	34.41	38.50
34	2.27	1.50	34.70	38.47
35	2.14	0.90	34.96	38.01

36	2.35	1.42	33.81	37.58
37	2.00	1.07	34.14	37.21
38	1.90	0.74	33.65	36.30
39	1.37	1.35	33.44	36.15
40	2.95	1.52	31.44	35.90
41	2.48	1.46	31.75	35.69
42	2.88	1.33	31.39	35.60
43	1.14	1.42	32.88	35.44
44	2.30	1.49	31.54	35.33
45	2.07	1.42	31.82	35.31
46	2.02	1.01	32.14	35.17
47	2.88	1.29	30.62	34.80
48	1.30	1.41	31.30	34.01
49	1.18	1.25	30.59	33.03
50	1.76	1.21	29.76	32.73
51	2.38	1.27	28.63	32.28
52	3.16	1.24	27.67	32.06
53	2.44	1.44	27.33	31.20
54	2.66	1.09	27.40	31.14
55	1.41	1.26	28.33	31.00
56	2.36	1.08	27.32	30.76
57	2.27	1.42	27.07	30.75
58	2.45	1.52	26.76	30.73
59	1.09	0.75	28.21	30.05
60	2.47	1.35	25.61	29.44
61	2.54	1.39	25.16	29.09
62	2.11	0.78	25.73	28.62
63	2.56	1.41	24.07	28.04
64	2.34	0.76	24.73	27.83
65	2.57	1.15	23.74	27.46
66	2.49	1.02	23.64	27.15
67	2.86	1.47	22.62	26.95
68	2.69	1.22	22.24	26.16
69	3.28	1.35	21.11	25.75
70	1.83	1.38	22.03	25.24
71	2.41	0.89	21.93	25.23
72	2.79	1.23	20.79	24.81
73	3.13	1.35	20.02	24.50
74	3.29	1.17	19.80	24.26

75	2.21	1.30	20.66	24.16
76	1.44	0.87	21.67	23.99
77	3.05	1.28	18.79	23.12
78	1.27	0.75	20.25	22.27
79	2.61	1.15	18.46	22.22
80	2.47	1.45	18.11	22.04
81	2.27	1.00	18.73	22.00
82	3.03	1.21	17.68	21.92
83	2.22	1.32	18.16	21.70
84	2.45	1.06	18.17	21.68
85	2.22	0.73	18.72	21.67
86	3.28	1.37	16.83	21.48
87	2.07	1.28	17.62	20.97
88	3.02	1.12	16.52	20.66
89	3.20	1.30	16.03	20.53
90	1.01	0.93	18.32	20.26
91	1.77	1.04	16.10	18.92
92	3.26	0.74	14.16	18.15
93	3.20	1.42	13.49	18.11
94	3.04	0.82	13.24	17.10
95	1.98	1.44	13.41	16.83
96	1.99	0.81	12.60	15.41
97	2.14	0.88	12.23	15.25
98	1.71	1.19	11.19	14.08
99	2.83	1.03	9.28	13.14
100	1.80	0.89	10.39	13.08

Optimized Approach

Tables 6.3 and Fig. 6.4 show the optimized end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for `QSet-1` (i.e., set of 100 queries which can be approximated using closed-forms) and `QSet-2` (i.e., set of 100 queries that can only be approximated using bootstrap), respectively.

Table 6.3: The optimized end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for (`QSet-1`). Each row represents a single query execution with a 10% error bound.

#	**Query Execution (s)**	**Error Estimation (s)**	**Diagnostics (s)**	**Total (s)**
1	1.07	0.30	1.11	2.47
2	1.15	0.31	1.01	2.47
3	1.05	0.26	1.15	2.47
4	1.18	0.23	1.06	2.47
5	1.31	0.15	0.99	2.46
6	1.09	0.37	1.00	2.46
7	0.98	0.29	1.19	2.46
8	0.99	0.25	1.21	2.45
9	1.15	0.21	1.09	2.45
10	0.86	0.32	1.27	2.45
11	1.11	0.16	1.18	2.45
12	1.04	0.32	1.08	2.44
13	1.11	0.36	0.97	2.44
14	1.22	0.28	0.93	2.43
15	1.39	0.29	0.76	2.43
16	1.02	0.24	1.17	2.43
17	1.11	0.25	1.06	2.42
18	1.06	0.35	1.02	2.42
19	1.18	0.23	1.00	2.41
20	1.05	0.33	1.02	2.41
21	1.16	0.17	1.07	2.40
22	1.23	0.24	0.93	2.40
23	1.01	0.36	1.03	2.40
24	1.09	0.24	1.07	2.39
25	0.87	0.22	1.30	2.38
26	1.16	0.26	0.96	2.38
27	1.02	0.34	1.01	2.38
28	1.03	0.15	1.19	2.37
29	1.02	0.33	1.02	2.36
30	0.88	0.31	1.18	2.36
31	1.20	0.18	0.96	2.34
32	1.21	0.14	1.00	2.34
33	1.08	0.26	1.00	2.34
34	1.06	0.27	1.02	2.34
35	0.96	0.34	1.04	2.34

36	1.17	0.29	0.88	2.33
37	1.14	0.29	0.91	2.33
38	0.91	0.28	1.14	2.33
39	0.95	0.19	1.19	2.33
40	0.98	0.20	1.15	2.32
41	1.12	0.19	1.02	2.32
42	0.81	0.24	1.27	2.32
43	0.91	0.22	1.19	2.32
44	0.72	0.29	1.31	2.31
45	1.08	0.33	0.90	2.31
46	1.06	0.38	0.87	2.31
47	1.06	0.32	0.92	2.30
48	0.92	0.36	1.01	2.29
49	1.23	0.19	0.87	2.29
50	0.88	0.33	1.08	2.29
51	1.09	0.23	0.96	2.28
52	0.94	0.28	1.06	2.28
53	0.82	0.21	1.24	2.27
54	0.97	0.35	0.95	2.26
55	0.87	0.35	1.03	2.26
56	1.00	0.20	1.06	2.25
57	1.05	0.25	0.95	2.25
58	1.16	0.13	0.94	2.24
59	0.86	0.19	1.18	2.23
60	1.09	0.23	0.91	2.23
61	0.98	0.19	1.04	2.22
62	0.83	0.32	1.04	2.19
63	0.82	0.38	0.98	2.18
64	0.96	0.20	1.02	2.17
65	0.96	0.31	0.89	2.17
66	0.84	0.27	1.05	2.16
67	0.79	0.27	1.10	2.16
68	0.77	0.18	1.21	2.15
69	1.02	0.19	0.93	2.15
70	0.98	0.32	0.86	2.15
71	0.86	0.19	1.09	2.14
72	0.87	0.15	1.08	2.09
73	0.71	0.35	0.99	2.06
74	0.86	0.18	1.02	2.05

75	0.85	0.16	1.04	2.05
76	0.82	0.31	0.92	2.04
77	0.81	0.24	0.98	2.04
78	0.86	0.34	0.84	2.03
79	0.86	0.14	1.02	2.02
80	0.79	0.33	0.90	2.02
81	0.76	0.15	1.10	2.01
82	0.55	0.15	1.29	1.99
83	0.83	0.22	0.94	1.99
84	0.92	0.15	0.89	1.96
85	0.79	0.14	1.03	1.96
86	0.85	0.22	0.89	1.96
87	0.99	0.22	0.75	1.95
88	0.69	0.25	1.00	1.94
89	0.89	0.20	0.84	1.93
90	0.86	0.17	0.89	1.92
91	0.85	0.19	0.88	1.92
92	0.82	0.37	0.69	1.88
93	0.53	0.37	0.96	1.86
94	0.87	0.24	0.74	1.85
95	0.79	0.30	0.73	1.82
96	0.80	0.16	0.85	1.81
97	0.78	0.14	0.86	1.77
98	0.81	0.16	0.76	1.73
99	0.97	0.16	0.59	1.72
100	0.83	0.16	0.63	1.61

Table 6.4: The optimized end-to-end response times and the individual overheads associated with query execution, error estimation, and diagnostics for (`QSet-2`). Each row represents a single query execution with a 10% error bound.

#	**Query Execution (s)**	**Error Estimation (s)**	**Diagnostics (s)**	**Total (s)**
1	0.87	0.10	1.43	2.40
2	0.82	0.15	1.33	2.30
3	0.80	0.15	1.33	2.29
4	0.72	0.16	1.40	2.28
5	0.82	0.17	1.23	2.21
6	0.72	0.13	1.33	2.18
7	0.78	0.16	1.21	2.15
8	0.61	0.11	1.41	2.14
9	0.76	0.08	1.29	2.13
10	0.60	0.13	1.37	2.10
11	0.64	0.17	1.28	2.10
12	0.75	0.16	1.17	2.08
13	0.82	0.17	1.07	2.06
14	0.71	0.08	1.25	2.05
15	0.70	0.13	1.22	2.05
16	0.70	0.11	1.24	2.05
17	0.65	0.16	1.23	2.05
18	0.72	0.13	1.19	2.04
19	0.73	0.11	1.20	2.04
20	0.76	0.09	1.18	2.03
21	0.74	0.14	1.14	2.02
22	0.58	0.11	1.33	2.02
23	0.81	0.17	1.02	2.01
24	0.78	0.17	1.05	2.00
25	0.80	0.09	1.11	2.00
26	0.55	0.11	1.33	1.99
27	0.64	0.14	1.21	1.99
28	0.81	0.11	1.07	1.99
29	0.78	0.11	1.10	1.99
30	0.52	0.15	1.30	1.97
31	0.62	0.16	1.18	1.97
32	0.64	0.09	1.23	1.96
33	0.67	0.11	1.18	1.96
34	0.79	0.13	1.04	1.96
35	0.69	0.16	1.10	1.95

36	0.44	0.12	1.39	1.95
37	0.55	0.10	1.28	1.94
38	0.76	0.14	1.03	1.92
39	0.57	0.15	1.20	1.92
40	0.69	0.12	1.11	1.91
41	0.57	0.08	1.27	1.91
42	0.82	0.10	0.99	1.90
43	0.61	0.11	1.17	1.89
44	0.74	0.17	0.99	1.89
45	0.76	0.08	1.02	1.86
46	0.69	0.09	1.08	1.85
47	0.32	0.12	1.42	1.85
48	0.52	0.17	1.15	1.84
49	0.59	0.17	1.08	1.84
50	0.62	0.13	1.08	1.83
51	0.59	0.14	1.10	1.83
52	0.71	0.09	1.03	1.83
53	0.64	0.08	1.10	1.83
54	0.57	0.10	1.15	1.82
55	0.44	0.15	1.22	1.82
56	0.61	0.13	1.07	1.81
57	0.59	0.15	1.07	1.81
58	0.57	0.10	1.13	1.79
59	0.55	0.08	1.15	1.78
60	0.66	0.11	1.00	1.77
61	0.63	0.16	0.98	1.77
62	0.55	0.17	1.04	1.76
63	0.50	0.16	1.10	1.76
64	0.59	0.09	1.07	1.75
65	0.49	0.08	1.18	1.75
66	0.55	0.16	1.02	1.74
67	0.62	0.16	0.95	1.73
68	0.62	0.15	0.96	1.73
69	0.44	0.11	1.17	1.73
70	0.54	0.15	1.03	1.72
71	0.50	0.13	1.09	1.72
72	0.65	0.08	0.99	1.72
73	0.61	0.11	1.00	1.72
74	0.60	0.13	0.99	1.71

75	0.28	0.17	1.26	1.71
76	0.46	0.13	1.11	1.70
77	0.53	0.15	1.02	1.70
78	0.55	0.10	1.04	1.70
79	0.43	0.17	1.10	1.70
80	0.53	0.09	1.07	1.69
81	0.59	0.09	0.99	1.67
82	0.53	0.09	1.03	1.65
83	0.25	0.15	1.24	1.65
84	0.52	0.15	0.97	1.64
85	0.50	0.13	1.00	1.64
86	0.36	0.15	1.12	1.63
87	0.27	0.15	1.20	1.62
88	0.41	0.17	1.04	1.62
89	0.50	0.11	1.00	1.61
90	0.45	0.12	1.04	1.60
91	0.47	0.15	0.96	1.58
92	0.48	0.12	0.97	1.57
93	0.34	0.12	1.10	1.56
94	0.32	0.16	1.03	1.51
95	0.35	0.15	0.97	1.47
96	0.40	0.10	0.95	1.46
97	0.34	0.11	0.98	1.42
98	0.28	0.14	0.99	1.42
99	0.31	0.09	1.01	1.40
100	0.30	0.08	0.98	1.36

Chapter 7

Conclusion

More and more organizations are increasingly turning towards extracting value from large volumes of data. In many cases, such value primarily comes from data-driven decisions. Hundreds of thousands of servers in data centers owned by corporations and businesses log millions of records every second. These records contain a variety of information — highly confidential financial or medical transactions, visitor information or even web content — and require analysts to run exploratory queries on huge volumes of data. For example, continuously analyzing large volumes of system logs can reveal critical performance bottlenecks in large-scale systems or analyzing user access patterns can give useful insights about what content is popular, which in turn can affect the ad placement strategies. While the individual use cases can be wide and varied, for many such scenarios, timeliness is more important than perfect accuracy, the queries are ad-hoc (i.e., they are not known in advance), and involve processing large volumes of data.

Achieving small, bounded response times for queries on these massive datasets has remained a challenge due to a number of reasons. These may range from limited disk bandwidth, growing datasets (and their inability to fit in memory), considerable network communication overhead during large data shuffles, or even straggling processes. For instance, executing a simple `GROUP BY` query on a few terabytes of data spread across hundreds of machines may take tens of minutes to just scan and process the data in parallel. This is often accompanied by unpredictable delays due to stragglers or network congestion during large data shuffles. Such delays severely impact the analyst's ability to carry out exploratory analysis on data.

In this thesis, we introduced BlinkDB [4, 5, 6], a massively parallel approximate-query processing framework optimized for interactive answers on large volumes of data. BlinkDB supports SPJA-style (i.e., select-project-join aggregate) SQL queries that can be annotated with either error or maximum execution time constraints, which the system uses for satisfying the user's requirements as well as for optimizing query execution.

BlinkDB accomplishes this by pre-computing and maintaining a carefully-chosen set of samples from the data, and executing queries on an appropriate sample that meets the error and time constraints of the query. To be able to handle queries that require samples from relatively infrequent sub-groups, BlinkDB maintains a set of *uniform* samples as well as several sets of *biased* samples, each *stratified* over a subset of columns.

Maintaining stratified samples for all subsets of the columns requires an exponential amount of storage and is hence, impractical. On the other hand, restricting stratified samples to only those subsets of columns that have appeared in past queries will limit the applicability of BlinkDB for ad-hoc queries. Therefore, we rely on an optimization framework to determine the set of columns on which to stratify, where the optimization formulation takes into account both the data distribution, past queries, storage constraints and several other system-related factors. Using these inputs, BlinkDB determines a set of samples which would help answer subsequent queries, while limiting additional storage used to a user configurable quantity.

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

Bibliography

[1] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. "Aqua: A Fast Decision Support Systems Using Approximate Query Answers". In: *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*. 1999, pp. 754–757. URL: http://www.vldb.org/conf/1999/P76.pdf.

[2] Swarup Acharya, Phillip B. Gibbons, and Viswanath Poosala. "Congressional Samples for Approximate Answering of Group-By Queries". In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*. 2000, pp. 487–498. DOI: 10.1145/342009.335450. URL: http://doi.acm.org/10.1145/342009.335450.

[3] Swarup Acharya et al. "The Aqua Approximate Query Answering System". In: *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. 1999, pp. 574–576. DOI: 10.1145/304182.304581. URL: http://doi.acm.org/10.1145/304182.304581.

[4] Sameer Agarwal et al. "Blink and It's Done: Interactive Queries on Very Large Data". In: *PVLDB* 5.12 (2012), pp. 1902–1905. URL: http://vldb.org/pvldb/vol5/p1902_sameeragarwal_vldb2012.pdf.

[5] Sameer Agarwal et al. "BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data". In: *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*. 2013, pp. 29–42. DOI: 10.1145/2465351.2465355. URL: http://doi.acm.org/10.1145/2465351.2465355.

[6] Sameer Agarwal et al. "Knowing when you're wrong: building fast and reliable approximate query processing systems". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. 2014, pp. 481–492. DOI: 10.1145/2588555.2593667. URL: http://doi.acm.org/10.1145/2588555.2593667.

[7] Sameer Agarwal et al. "Reoptimizing Data Parallel Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. 2012, pp. 281–294. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/agarwal.

[8] Charu C. Aggarwal. "On Biased Reservoir Sampling in the Presence of Stream Evolution". In: *Proceedings of the 32Nd International Conference on Very Large Data Bases*. VLDB '06. Seoul, Korea: VLDB Endowment, 2006, pp. 607–618. URL: http://dl.acm.org/citation.cfm?id=1182635.1164180.

[9] Parag Agrawal, Daniel Kifer, and Christopher Olston. "Scheduling Shared Scans of Large Data Files". In: *Proc. VLDB Endow.* 1.1 (Aug. 2008), pp. 958–969. ISSN: 2150-8097. DOI: 10.14778/1453856.1453960. URL: http://dx.doi.org/10.14778/1453856.1453960.

[10] *AMPLab Big Data Benchmark*. https://amplab.cs.berkeley.edu/benchmark/.

[11] Ganesh Ananthanarayanan et al. "Reining in the Outliers in Map-Reduce Clusters using Mantri". In: *9th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2010, October 4-6, 2010, Vancouver, BC, Canada, Proceedings*. 2010, pp. 265–278. URL: http://www.usenix.org/events/osdi10/tech/full_papers/Ananthanarayanan.pdf.

[12] Ganesh Ananthanarayanan et al. "Scarlett: coping with skewed content popularity in mapreduce clusters". In: *European Conference on Computer Systems, Proceedings of the Sixth European conference on Computer systems, EuroSys 2011, Salzburg, Austria, April 10-13, 2011*. 2011, pp. 287–300. DOI: 10.1145/1966445.1966472. URL: http://doi.acm.org/10.1145/1966445.1966472.

[13] Alexandr Andoni, Robert Krauthgamer, and Krzysztof Onak. "Streaming Algorithms from Precision Sampling". In: *CoRR* abs/1011.1263 (2010). URL: http://arxiv.org/abs/1011.1263.

[14] Gennady Antoshenkov. "Random Sampling from Pseudo-Ranked B+ Trees". In: *Proceedings of the 18th International Conference on Very Large Data Bases*. VLDB '92. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992, pp. 375–382. ISBN: 1-55860-151-1. URL: http://dl.acm.org/citation.cfm?id=645918.672499.

[15] *Apache Hadoop MapReduce Project*. http://hadoop.apache.org/mapreduce/.

[16] Ron Avnur and Joseph M. Hellerstein. "Eddies: Continuously Adaptive Query Processing". In: *SIGMOD Rec.* 29.2 (May 2000), pp. 261–272. ISSN: 0163-5808. DOI: 10.1145/335191.335420. URL: http://doi.acm.org/10.1145/335191.335420.

[17] Brian Babcock, Surajit Chaudhuri, and Gautam Das. "Dynamic Sample Selection for Approximate Query Processing". In: *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. SIGMOD '03. San Diego, California: ACM, 2003, pp. 539–550. ISBN: 1-58113-634-X. DOI: 10.1145/872757.872822. URL: http://doi.acm.org/10.1145/872757.872822.

[18] Brian Babcock, Mayur Datar, and Rajeev Motwani. "Sampling from a Moving Window over Streaming Data". In: *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA '02. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 633–634. ISBN: 0-89871-513-X. URL: http://dl.acm.org/citation.cfm?id=545381.545465.

[19] Linas Baltrunas, Arturas Mazeika, and Michael Bohlen. "Multi-dimensional Histograms with Tight Bounds for the Error". In: *Proceedings of the 10th International Database Engineering and Applications Symposium*. IDEAS '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 105–112. ISBN: 0-7695-2577-6. DOI: 10.1109/IDEAS.2006.31. URL: http://dx.doi.org/10.1109/IDEAS.2006.31.

[20] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. "STHoles: A Multidimensional Workload-aware Histogram". In: *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*. SIGMOD '01. Santa Barbara, California, USA: ACM, 2001, pp. 211–222. ISBN: 1-58113-332-4. DOI: 10.1145/375663.375686. URL: http://doi.acm.org/10.1145/375663.375686.

[21] Nicolas Bruno et al. "Recurring Job Optimization in Scope". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. SIGMOD '12. Scottsdale, Arizona, USA: ACM, 2012, pp. 805–806. ISBN: 978-1-4503-1247-9. DOI: 10.1145/2213836.2213959. URL: http://doi.acm.org/10.1145/2213836.2213959.

[22] Francesco Buccafurri et al. "Binary-Tree Histograms with Tree Indices". In: *Proceedings of the 13th International Conference on Database and Expert Systems Applications*. DEXA '02. London, UK, UK: Springer-Verlag, 2002, pp. 861–870. ISBN: 3-540-44126-3. URL: http://dl.acm.org/citation.cfm?id=648315.756194.

[23] Francesco Buccafurri et al. "Enhancing histograms by tree-like bucket indices". In: *VLDB J.* 17.5 (2008), pp. 1041–1061. DOI: 10.1007/s00778-007-0050-5. URL: http://dx.doi.org/10.1007/s00778-007-0050-5.

[24] Chiranjeeb Buragohain, Nisheeth Shrivastava, and Subhash Suri. "Space Efficient Streaming Algorithms for the Maximum Error Histogram". In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)* 0 (2007), pp. 1026–1035. DOI: http://doi.ieeecomputersociety.org/10.1109/ICDE.2007.368961.

[25] Michael Cafarella et al. "Data management projects at Google". In: *SIGMOD Rec.* 37.1 (Mar. 2008), pp. 34–38. ISSN: 0163-5808. DOI: 10.1145/1374780.1374789. URL: http://doi.acm.org/10.1145/1374780.1374789.

[26] Angelo J. Canty et al. "Bootstrap diagnostics and remedies". In: *Canadian Journal of Statistics* 34.1 (2006), pp. 5–27. ISSN: 1708-945X. DOI: 10.1002/cjs.5550340103. URL: http://dx.doi.org/10.1002/cjs.5550340103.

[27] Kaushik Chakrabarti et al. "Approximate Query Processing Using Wavelets". In: *Proceedings of the 26th International Conference on Very Large Data Bases*. VLDB '00. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 111–122. ISBN: 1-55860-715-3. URL: http://dl.acm.org/citation.cfm?id=645926.671851.

[28] Surajit Chaudhuri, Gautam Das, and Vivek Narasayya. "Optimized Stratified Sampling for Approximate Query Processing". In: *ACM Trans. Database Syst.* 32.2 (June 2007). ISSN: 0362-5915. DOI: 10.1145/1242524.1242526. URL: http://doi.acm.org/10.1145/1242524.1242526.

[29] Surajit Chaudhuri, Gautam Das, and Utkarsh Srivastava. "Effective Use of Block-level Sampling in Statistics Estimation". In: *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*. SIGMOD '04. Paris, France: ACM, 2004, pp. 287–298. ISBN: 1-58113-859-8. DOI: 10.1145/1007568.1007602. URL: http://doi.acm.org/10.1145/1007568.1007602.

[30] Surajit Chaudhuri and Umeshwar Dayal. "An Overview of Data Warehousing and OLAP Technology". In: *SIGMOD Rec.* 26.1 (Mar. 1997), pp. 65–74. ISSN: 0163-5808. DOI: 10.1145/248603.248616. URL: http://doi.acm.org/10.1145/248603.248616.

[31] Surajit Chaudhuri and Vivek Narasayya. "AutoAdmin &Ldquo;What-if&Rdquo; Index Analysis Utility". In: *SIGMOD Rec.* 27.2 (June 1998), pp. 367–378. ISSN: 0163-5808. DOI: 10.1145/276305.276337. URL: http://doi.acm.org/10.1145/276305.276337.

[32] Mosharaf Chowdhury et al. "Managing data transfers in computer clusters with orchestra". In: *Proceedings of the ACM SIGCOMM 2011 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Toronto, ON, Canada, August 15-19, 2011*. 2011, pp. 98–109. DOI: 10.1145/2018436.2018448. URL: http://doi.acm.org/10.1145/2018436.2018448.

[33] Edith Cohen, Graham Cormode, and Nick Duffield. "Structure-aware Sampling on Data Streams". In: *Proceedings of the ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems*. SIGMETRICS '11. San Jose, California, USA: ACM, 2011, pp. 197–208. ISBN: 978-1-4503-0814-4. DOI: 10.1145/1993744.1993763. URL: http://doi.acm.org/10.1145/1993744.1993763.

[34] *Common Crawl Document Corpus*. http://commoncrawl.org/.

[35] Tyson Condie et al. "MapReduce Online". In: *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2010, April 28-30, 2010, San Jose, CA, USA*. 2010, pp. 313–328. URL: http://www.usenix.org/events/nsdi10/tech/full_papers/condie.pdf.

[36] *Conviva Inc.* http://www.conviva.com/.

[37] Graham Cormode. "Sketch Techniques for Massive Data". In: *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*. Ed. by Graham Cormode et al. Foundations and Trends in Databases. NOW publishers, 2011.

[38] Graham Cormode et al. "Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches". In: *Found. Trends databases* 4 (Jan. 2012), pp. 1–294. ISSN: 1931-7883. DOI: 10.1561/1900000004. URL: http://dx.doi.org/10.1561/1900000004.

[39] Alin Dobra et al. "Turbo-Charging Estimate Convergence in DBO". In: *PVLDB* 2.1 (2009), pp. 419–430. URL: http://www.vldb.org/pvldb/2/vldb09-234.pdf.

[40] Bradley Efron. "Bootstrap methods: another look at the jackknife". In: *The annals of Statistics* (1979), pp. 1–26.

[41] Bradley Efron and Robert Tibshirani. *An introduction to the bootstrap*. Vol. 57. CRC press, 1993.

[42] *Facebook Presto*. http://prestodb.io/.

[43] Minos N. Garofalakis and Phillip B. Gibbons. "Approximate Query Processing: Taming the TeraBytes". In: *VLDB 2001, Proceedings of 27th International Conference on Very Large Data Bases, September 11-14, 2001, Roma, Italy*. 2001. URL: http://www.vldb.org/conf/2001/tut4.pdf.

[44] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. "SharedDB: Killing One Thousand Queries With One Stone". In: *PVLDB* 5.6 (2012), pp. 526–537. URL: http://vldb.org/pvldb/vol5/p526_georgiosgiannikis_vldb2012.pdf.

[45] Peter J. Haas and Peter J. Haas. "Hoeffding Inequalities for Join-Selectivity Estimation and Online Aggregation". In: *IBM Research Report RJ 10040, IBM Almaden Research*. 1996.

[46] Peter Hall. "On Symmetric Bootstrap Confidence Intervals". English. In: *Journal of the Royal Statistical Society. Series B (Methodological)* 50.1 (1988), pp. 35–45. ISSN: 00359246. URL: http://www.jstor.org/stable/2345806.

[47] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. "Online Aggregation". In: *SIGMOD 1997, Proceedings ACM SIGMOD International Conference on Management of Data, May 13-15, 1997, Tucson, Arizona, USA*. 1997, pp. 171–182. DOI: 10.1145/253260.253291. URL: http://doi.acm.org/10.1145/253260.253291.

[48] Herodotos Herodotou et al. "Starfish: A Self-tuning System for Big Data Analytics". In: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. 2011, pp. 261–272. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper36.pdf.

[49] Benjamin Hindman et al. "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. 2011. URL: https://www.usenix.org/conference/nsdi11/mesos-platform-fine-grained-resource-sharing-data-center.

[50] *Intel Hadoop Benchmark*. https://github.com/intel-hadoop/HiBench.

[51] Navin Kabra and David J. DeWitt. "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans". In: *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*. 1998, pp. 106–117. DOI: 10.1145/276304.276315. URL: http://doi.acm.org/10.1145/276304.276315.

[52] Ariel Kleiner et al. "A general bootstrap performance diagnostic". In: *The 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, KDD 2013, Chicago, IL, USA, August 11-14, 2013*. 2013, pp. 419–427. DOI: 10.1145/2487575.2487650. URL: http://doi.acm.org/10.1145/2487575.2487650.

[53] Nikolay Laptev, Kai Zeng, and Carlo Zaniolo. "Early Accurate Results for Advanced Analytics on MapReduce". In: *PVLDB* 5.10 (2012), pp. 1028–1039. URL: http://vldb.org/pvldb/vol5/p1028_nikolaylaptev_vldb2012.pdf.

[54] Xiaolei Li et al. "Sampling Cube: A Framework for Statistical Olap over Sampling Data". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. SIGMOD '08. Vancouver, Canada: ACM, 2008, pp. 779–790. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376695. URL: http://doi.acm.org/10.1145/1376616.1376695.

[55] S.L. Lohr. *Sampling: design and analysis*. Thomson, 2009.

[56] Colin McDiarmid. "Concentration". In: *Probabilistic Methods for Algorithmic Discrete Mathematics*. Vol. 16. Algorithms and Combinatorics. Springer Berlin Heidelberg, 1998, pp. 195–248. ISBN: 978-3-642-08426-3. DOI: 10.1007/978-3-662-12788-9_6. URL: http://dx.doi.org/10.1007/978-3-662-12788-9_6.

[57] Sergey Melnik et al. "Dremel: interactive analysis of web-scale datasets". In: *Commun. ACM* 54 (6 June 2011), pp. 114–123. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/1953122.1953148. URL: http://doi.acm.org/10.1145/1953122.1953148.

[58] Robert B. Miller. "Response Time in Man-computer Conversational Transactions". In: *Proceedings of the December 9-11, 1968, Fall Joint Computer Conference, Part I*. AFIPS '68 (Fall, part I). San Francisco, California: ACM, 1968, pp. 267–277. DOI: 10.1145/1476589.1476628. URL: http://doi.acm.org/10.1145/1476589.1476628.

[59] Barzan Mozafari and Carlo Zaniolo. "Optimal load shedding with aggregates and mining queries". In: *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. 2010, pp. 76–88. DOI: 10.1109/ICDE.2010.5447867. URL: http://dx.doi.org/10.1109/ICDE.2010.5447867.

[60] Derek Gordon Murray et al. "CIEL: A Universal Execution Engine for Distributed Data-Flow Computing". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. 2011. URL: https://www.usenix.org/conference/nsdi11/ciel-universal-execution-engine-distributed-data-flow-computing.

[61] *Netflix Inc.* http://www.netflix.com/.

[62] Frank Olken and Doron Rotem. "Simple Random Sampling from Relational Databases". In: *Proceedings of the 12th International Conference on Very Large Data Bases*. VLDB '86. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1986, pp. 160–169. ISBN: 0-934613-18-4. URL: http://dl.acm.org/citation.cfm?id=645913.671474.

[63] Christopher Olston et al. "Interactive Analysis of Web-Scale Data". In: *CIDR 2009, Fourth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 4-7, 2009, Online Proceedings*. 2009. URL: http://www-db.cs.wisc.edu/cidr/cidr2009/Paper_21.pdf.

[64] Nikunj C. Oza. "Online bagging and boosting". In: *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics, Waikoloa, Hawaii, USA, October 10-12, 2005*. 2005, pp. 2340–2345. DOI: 10.1109/ICSMC.2005.1571498. URL: http://dx.doi.org/10.1109/ICSMC.2005.1571498.

[65] Niketan Pansare et al. "Online Aggregation for Large MapReduce Jobs". In: *PVLDB* 4.11 (2011), pp. 1135–1145. URL: http://www.vldb.org/pvldb/vol4/p1135-pansare.pdf.

[66] Abhijit Pol and Chris Jermaine. "Relational Confidence Bounds Are Easy With The Bootstrap". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, Baltimore, Maryland, USA, June 14-16, 2005*. 2005, pp. 587–598. DOI: 10.1145/1066157.1066224. URL: http://doi.acm.org/10.1145/1066157.1066224.

[67] *Putting a Bolder Face on Google (NYTimes)*. http://www.nytimes.com/2009/03/01/business/01marissa.html.

[68] Chengjie Qin and Florin Rusu. "PF-OLA: a high-performance framework for parallel online aggregation". English. In: *Distributed and Parallel Databases* (2013), pp. 1–39. ISSN: 0926-8782. DOI: 10.1007/s10619-013-7132-8. URL: http://dx.doi.org/10.1007/s10619-013-7132-8.

[69] *Qubole: Fast Query Authoring Tools*. http://www.qubole.com/s/overview.

[70] J.A. Rice. *Mathematical Statistics and Data Analysis*. Brooks/Cole Pub. Co., 1988. ISBN: 9781111793715. URL: http://books.google.com/books?id=7SI8AAAAQBAJ.

[71] Carsten Sapia. "PROMISE: Predicting Query Behavior to Enable Predictive Caching Strategies for OLAP Systems". In: *Data Warehousing and Knowledge Discovery, Second International Conference, DaWaK 2000, London, UK, September 4-6, 2000, Proceedings*. 2000, pp. 224–233. DOI: 10.1007/3-540-44466-1_22. URL: http://dx.doi.org/10.1007/3-540-44466-1_22.

[72] Alan Shieh et al. "Sharing the Data Center Network". In: *Proceedings of the 8th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2011, Boston, MA, USA, March 30 - April 1, 2011*. 2011. URL: https://www.usenix.org/conference/nsdi11/sharing-data-center-network.

[73] Lefteris Sidirourgos, Martin L. Kersten, and Peter A. Boncz. "SciBORQ: Scientific data management with Bounds On Runtime and Quality". In: *CIDR 2011, Fifth Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 9-12, 2011, Online Proceedings*. 2011, pp. 296–301. URL: http://www.cidrdb.org/cidr2011/Papers/CIDR11_Paper39.pdf.

[74] Ashish Thusoo et al. "Hive - A Warehousing Solution Over a Map-Reduce Framework". In: *PVLDB* 2.2 (2009), pp. 1626–1629. URL: http://www.vldb.org/pvldb/2/vldb09-938.pdf.

[75] *TPC-H Query Processing Benchmarks*. http://www.tpc.org/tpch/.

[76] *Twitter Inc.* http://www.twitter.com/.

[77] A.W. van der Vaart. *Asymptotic statistics*. Vol. 3. Cambridge university press, 2000.

[78] A.W. van der Vaart and J. Wellner. *Weak Convergence and Empirical Processes: With Applications to Statistics*. Springer Series in Statistics. Springer, 1996. ISBN: 9780387946405. URL: http://books.google.com/books?id=seH8dMrEgggC.

[79] Jeffrey Scott Vitter and Min Wang. "Approximate Computation of Multidimensional Aggregates of Sparse Data Using Wavelets". In: *SIGMOD 1999, Proceedings ACM SIGMOD International Conference on Management of Data, June 1-3, 1999, Philadelphia, Pennsylvania, USA*. 1999, pp. 193–204. DOI: 10.1145/304182.304199. URL: http://doi.acm.org/10.1145/304182.304199.

[80] Xiaodan Wang et al. "CoScan: cooperative scan sharing in the cloud". In: *ACM Symposium on Cloud Computing in conjunction with SOSP 2011, SOCC '11, Cascais, Portugal, October 26-28, 2011*. 2011, p. 11. DOI: 10.1145/2038916.2038927. URL: http://doi.acm.org/10.1145/2038916.2038927.

[81] Reynold S. Xin et al. "Shark: SQL and rich analytics at scale". In: *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*. 2013, pp. 13–24. DOI: 10.1145/2463676.2465288. URL: http://doi.acm.org/10.1145/2463676.2465288.

[82] Matei Zaharia et al. "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling". In: *European Conference on Computer Systems, Proceedings of the 5th European conference on Computer systems, EuroSys 2010, Paris, France, April 13-16, 2010*. 2010, pp. 265–278. DOI: 10.1145/1755913.1755940. URL: http://doi.acm.org/10.1145/1755913.1755940.

[83] Matei Zaharia et al. "Improving MapReduce Performance in Heterogeneous Environments". In: *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 2008, pp. 29–42. URL: http://www.usenix.org/events/osdi08/tech/full_papers/zaharia/zaharia.pdf.

[84] Matei Zaharia et al. "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing". In: *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2012, San Jose, CA, USA, April 25-27, 2012*. 2012, pp. 15–28. URL: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia.

[85] Kai Zeng et al. "The analytical bootstrap: a new method for fast error estimation in approximate query processing". In: *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*. 2014, pp. 277–288. DOI: 10.1145/2588555.2588579. URL: http://doi.acm.org/10.1145/2588555.2588579.