BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data

Sameer Agarwal†, Barzan Mozafari°, Aurojit Panda†, Henry Milner†, Samuel Madden°, Ion Stoica*†

†*University of California, Berkeley* °*Massachusetts Institute of Technology* **Conviva Inc.*
{sameerag, apanda, henrym, istoica}@cs.berkeley.edu, {barzan, madden}@csail.mit.edu

Abstract

In this paper, 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 two key ideas: (1) an adaptive optimization framework that builds and maintains a set of multi-dimensional stratified samples from original data over time, and (2) a dynamic sample selection strategy that selects an appropriately sized sample based on a query's accuracy or response time requirements. We evaluate BlinkDB against the well-known TPC-H benchmarks and a real-world analytic workload derived from Conviva Inc., a company that manages video distribution over the Internet. Our experiments on a 100 node cluster show that BlinkDB can answer queries on up to 17 TBs of data in less than 2 seconds (over 200× faster than Hive), within an error of 2-10%.

1. Introduction

Modern data analytics applications involve computing aggregates over a large number of records to *roll-up* web clicks, online transactions, content downloads, and other features along a variety of different dimensions, including demographics, content type, region, and so on. Traditionally, such queries have been executed using sequential scans over a large fraction of a database. Increasingly, new applications demand near real-time response rates. Examples may include applications that (i) update ads on a website based on trends in social networks like Facebook and Twitter, or (ii) determine the subset of users experiencing poor performance based on their service provider and/or geographic location.

Over the past two decades a large number of approximation techniques 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 [10, 14], sketches [12], and on-line aggregation [15]. 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 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 [16].

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 sampling and sketch based solutions 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 maintain random samples that omit data about smaller cities.

At the other end of the spectrum, systems like online aggregation (OLA) [15] make fewer assumptions about the query workload, at the expense of highly variable performance. Using OLA, 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, OLA may need to read the entire table to compute a result with satisfactory error bounds.

In this paper, we argue that none of the previous solutions are a good fit for today's big data analytics workloads. OLA

Eurosys'13 April 15-17, 2013, Prague, Czech Republic

provides relatively poor performance for queries on rare tuples, while sampling and sketches make strong assumptions about the predictability of workloads or substantially limit the types of queries they can execute.

To this end, we propose BlinkDB, a distributed sampling-based approximate query processing system that strives to achieve a better balance between efficiency and generality for analytics workloads. BlinkDB allows users to pose SQL-based aggregation queries 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 most existing approximate query solutions (*e.g.,* [10]), 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 paper.

BlinkDB consists of two main modules: (i) Sample Creation and (ii) Sample Selection. 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 formulate 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 show 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. This is in contrast to previous optimization-based sampling systems that assume complete knowledge of the tuples accessed by queries at optimization time.

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. It uses an *Error-Latency Profile* heuristic to efficiently choose the sample that will best satisfy the user-specified error or time bounds.

We implemented BlinkDB[1] on top of Hive/Hadoop [22] (as well as Shark [13], an optimized Hive/Hadoop framework that caches input/ intermediate data). Our implementation requires minimal changes to the underlying query processing system. We validate its effectiveness on a 100 node cluster, using both the TPC-H benchmarks and a real-world workload derived from Conviva. Our experiments show that BlinkDB can answer a range of queries within 2 seconds on 17 TB of data within 90-98% accuracy, which is two orders of magnitude faster than running the same queries on Hive/Hadoop. In summary, we make the following contributions:

- We use a column-set based optimization framework to compute a set of stratified samples (in contrast to approaches like AQUA [6] and STRAT [10], which compute only a single sample per table). Our optimization takes into account: (i) the frequency of rare subgroups in the data, (ii) the column sets in the past queries, and (iii) the storage overhead of each sample. (§4)
- We create *error-latency profiles (ELPs)* for each query at runtime to estimate its error or response time on each available sample. This heuristic is then used to select the most appropriate sample to meet the query's response time or accuracy requirements. (§5)
- We show how to integrate our approach into an existing parallel query processing framework (Hive) with minimal changes. We demonstrate that by combining these ideas together, 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. (§6)

2. Background

Any 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 BlinkDB and 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.

2.1 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

[1] http://blinkdb.org

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. 1, we classify possible approaches into one of four categories:

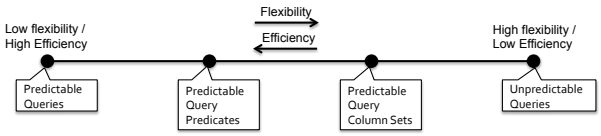


Figure 1. Taxonomy of workload models.

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 [12] 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) [14].

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 [10] and SciBORQ [21], 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.

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 [20] has shown that a similar model can be used to improve caching performance in OLAP systems. AQUA [4], an approximate query database based on sampling, uses the QCS model. (See §8 for a comparison between AQUA and BlinkDB).

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 On-Line Aggregation (OLA) [15], which relies on streaming data in random order.

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 [11, 19] 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 at the HDFS block[2] 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.

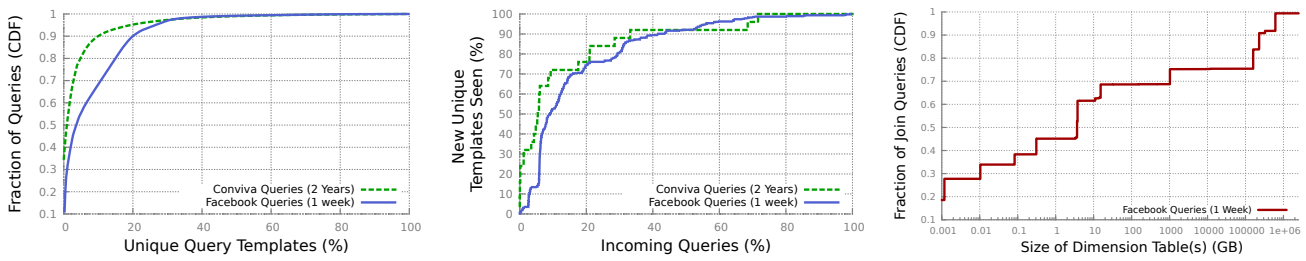
2.2 Query Patterns in a Production Cluster

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 7 days' workload from Facebook to determine the frequency of QCSs.

Fig. 2(a) 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(b) 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 Face-

[2] Typically, these blocks are 64 – 1024 MB in size.



(a) QCS Distribution (b) QCS Stability (c) Dimension table size CDF

Figure 2. 2(a) and 2(b) show the distribution and stability of QCSs respectively across all queries in the Conviva and Facebook traces. 2(c) shows the distribution of join queries with respect to the size of dimension tables.

book 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.

3. System Overview

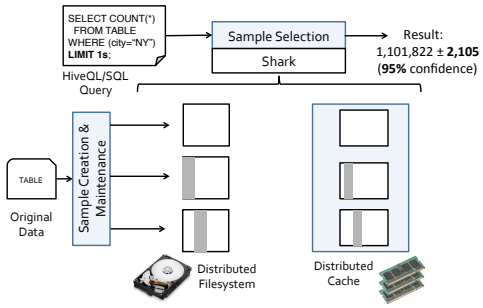


Figure 3. BlinkDB architecture.

Fig. 3 shows the overall architecture of BlinkDB. BlinkDB extends the Apache Hive framework [22] by adding two major components to it: (1) an offline sampling module that creates and maintains samples over time, and (2) a run-time sample selection module that creates an *Error-Latency Profile (ELP)* for queries. To decide on the samples to create, we use the QCSs that appear in queries (we present a more precise formulation of this mechanism in §4.) Once this choice is made, we rely on distributed reservoir sampling[3] [23] or binomial sampling techniques to create a range of uniform and stratified samples across a number of dimensions.

At run-time, we employ ELP to decide the sample to run the query. The ELP characterizes the rate at which the error (or response time) decreases (or increases) as the size of the sample on which the query operates increases. This is used to select a sample that best satisfies the user's constraints. We describe ELP in detail in §5. BlinkDB also augments the query parser, optimizer, and a number of aggregation operators to allow queries to specify bounds on error, or execution time.

[3] Reservoir sampling is a family of randomized algorithms for creating fixed-sized random samples from streaming data.

3.1 Supported Queries

BlinkDB supports a slightly constrained set of SQL-style declarative queries, imposing constraints that are similar to prior work [10]. In particular, BlinkDB can currently provide approximate results for standard SQL aggregate queries involving `COUNT`, `AVG`, `SUM` and `QUANTILE`. Queries involving these operations can be annotated with either an error bound, or a time constraint. Based on these constraints, the system selects an appropriate sample, of an appropriate size, as explained in §5.

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%
```

will return the count for each `GROUP BY` key, with each count having relative error of at most ±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
```

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.

While BlinkDB does not currently support arbitrary joins and nested SQL queries, we find that this is usually not a hindrance. This is because any query involving nested queries or joins can be flattened to run on the underlying data. However, we do provide support for joins in some settings which are commonly used in distributed data warehouses. In particular, BlinkDB can support joining a large, sampled fact table, with smaller tables that are small enough to fit in the main memory of any single node in the cluster. This is one

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 1. Notation in §4.1

of the most commonly used form of joins in distributed data warehouses. For instance, Fig. 2(c) shows the distribution of the size of *dimension tables* (*i.e.,* all tables except the largest) across all queries in a week's trace from Facebook. We observe that 70% of the queries involve dimension tables that are less than 100 GB in size. These dimension tables can be easily cached in the cluster memory, assuming a cluster consisting of hundreds or thousands of nodes, where each node has at least 32 GB RAM. It would also be straightforward to extend BlinkDB to deal with foreign key joins between two sampled tables (or a self join on one sampled table) where both tables have a stratified sample on the set of columns used for joins. We are also working on extending our query model to support more general queries, specifically focusing on more complicated user defined functions, and on nested queries.

4. 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, in §4.1, we discuss the creation of a stratified sample on a given set of columns. We show how a query's 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. In §4.2 we formulate and solve an optimization problem to decide on the sets of columns on which we build samples.

4.1 Stratified Samples

In this section, we describe our techniques for constructing a sample to target queries using a given QCS. Table 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* [16], which ensures that rare subgroups are sufficiently represented. Next, we describe the use of stratified sampling in BlinkDB.

4.1.1 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 sample size on which we can operate, n; n is also the optimal sample size, since larger samples produce better statistical results. 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 (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 §5 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, $T_x = \{r : r \in T \text{ and } r \text{ takes values } x \text{ on columns } \phi\}$. 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 $|S_x|$ 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. [4]. This leads to the following algorithm for sample selection:

1. **Compute group counts:** To each $x \in x_0, \ldots, x_{|D(\phi)|-1}$, assign a count, forming a $|D(\phi)|$-vector of counts N_n^*. Compute N_n^* as follows: Let $N(n') = (\min(\lfloor \frac{n'}{|D(\phi)|} \rfloor, |T_{x_0}|), \min(\lfloor \frac{n'}{|D(\phi)|} \rfloor, |T_{x_1}|, \ldots)$, 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.

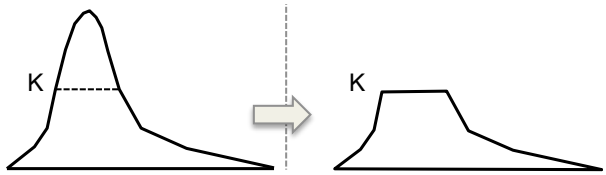


Figure 4. Example of a stratified sample associated with a set of columns, ϕ.

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 $S(\phi, K)$ 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 $\sqrt{K}$ [16].

4.1.2 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. 5(a) 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.

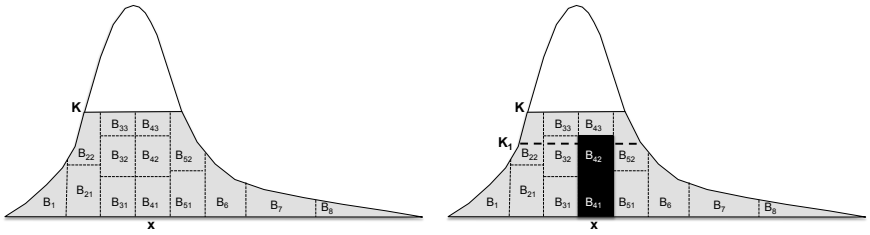
When S_x is spread over multiple blocks, each block contains a randomly ordered random subset from S_x, and, by extension, from the original table. This makes it possible to efficiently run queries on smaller samples. Assume a query Q, that needs to read n rows in total to satisfy its error bounds or 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. 5(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%.

4.2 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 [9] or on a single multi-dimensional (*i.e.,* multi-column) stratified sample [4], BlinkDB creates several multi-



(a) (b)

Figure 5. (a) Possible storage layout for stratified sample $S(\phi, K)$.

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.

4.2.1 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.

[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.

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:

$$G = \sum_j p_j \cdot y_j \cdot \Delta(q_j, M) \quad (1)$$

subject to

$$\sum_{i=1}^{m} |S(\phi_i, K)| \cdot z_i \leq \mathbb{C} \quad (2)$$

and

$$\forall\ j: \quad y_j \leq \max_{i:\phi_i \subseteq q_j \cup i:\phi_i \supset q_j} \left(z_i \min 1, \frac{|D(\phi_i)|}{|D(q_j)|}\right) \quad (3)$$

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 (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 a the QCS for a query, a partially-covering QCS may be used instead. In particular, the uniform sample is a degenerate case with $\phi_i = \varnothing$; 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 paper we approximate it by y_j, which is determined by constraint (3). The y_j value is in $[0, 1]$, with 0 meaning no coverage, and 1 meaning full coverage. The intuition behind (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 (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 §7 show that the system performs quite well on the datasets we consider using these parameter values.

5. BlinkDB Runtime

In this section, we provide an overview of query execution in BlinkDB and present our approach for online sample selection. 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 underlying 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 section, we detail our approach to query execution, by first discussing our mechanism for selecting a set of appropriate samples (§5.1), and then picking an appropriate subset size from one of those samples by constructing the *Error Latency Profile* for the query (§5.2). Finally, we discuss how BlinkDB corrects the bias introduced by executing queries on stratified samples (§5.4).

5.1 Selecting the 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.

5.2 Selecting the Right 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 [16]. 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 [16]. 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 [8, 26]. 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.

5.3 An Example

As an illustrative example consider a query which calculates average session time for "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 media area 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 6. 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.

5.4 Bias Correction

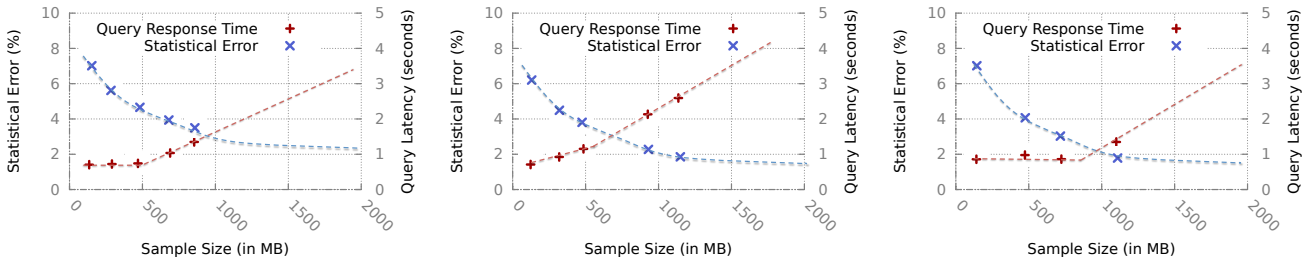
Running a query on a non-uniform sample introduces a certain amount of statistical bias in the final result since different groups are picked at different frequencies. In particular while all the tuples matching a rare subgroup would be included in the sample, more popular subgroups will only have a 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.

6. Implementation

Fig. 7 describes the entire BlinkDB ecosystem. BlinkDB is built on top of the Hive Query Engine [22], supports both Hadoop MapReduce [2] and Spark [25] (via Shark [13]) at the execution layer and uses the Hadoop Distributed File System [1] at the storage layer.

Our implementation required changes in a few key components. We add a shim layer to the HiveQL parser to handle the *BlinkDB Query Interface*, which enables queries with response time and error bounds. Furthermore, the query interface can detect data input, triggering the *Sample Creation and Maintenance* module, which creates or updates the set of random and multi-dimensional samples as described in §4. We further extend the HiveQL parser to implement a *Sample Selection* module that re-writes the query and iteratively assigns it an appropriately sized uniform or stratified sample as described in §5. We also add an *Uncertainty Propagation* module to modify all pre-existing aggregation functions with statistical closed forms to return errors bars and confidence intervals in addition to the result.

One concern with BlinkDB is that multiple queries might use the same sample, inducing correlation among the answers to those queries. For example, if by chance a sample has a higher-than-expected average value of an aggregation column, then two queries that use that sample and aggregate on that column will both return high answers. This may



(a) `dt, country` (b) `dt, dma` (c) `dt, ended_flag`

Figure 6. 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`.

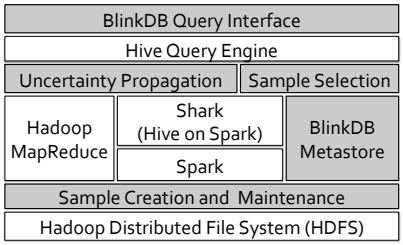


Figure 7. BlinkDB's Implementation Stack

introduce subtle inaccuracies in analysis based on multiple queries. By contrast, in a system that creates a new sample for each query, a high answer for the first query is not predictive of a high answer for the second. However, as we have already discussed in §2, precomputing samples is essential for performance in a distributed setting. We address correlation among query results by periodically replacing the set of samples used. BlinkDB runs a low priority background task which periodically (typically, daily) samples from the original data, creating new samples which are then used by the system.

An additional concern is that the workload might change over time, and the sample types we compute are no longer "optimal". To alleviate this concern, BlinkDB keeps track of statistical properties of the underlying data (*e.g.,* variance and percentiles) and periodically runs the sample creation module described in §4 to re-compute these properties and decide whether the set of samples needs to be changed. To reduce the churn caused due to this process, an operator can set a parameter to control the percentage of sample that can be changed at any single time.

In BlinkDB, uniform samples are generally created in a few hundred seconds. This is because the time taken to create them only depends on the disk/memory bandwidth and the degree of parallelism. On the other hand, creating stratified samples on a set of columns takes anywhere between a 5 – 30 minutes depending on the number of unique values to stratify on, which decides the number of reducers and the amount of data shuffled.

7. Evaluation

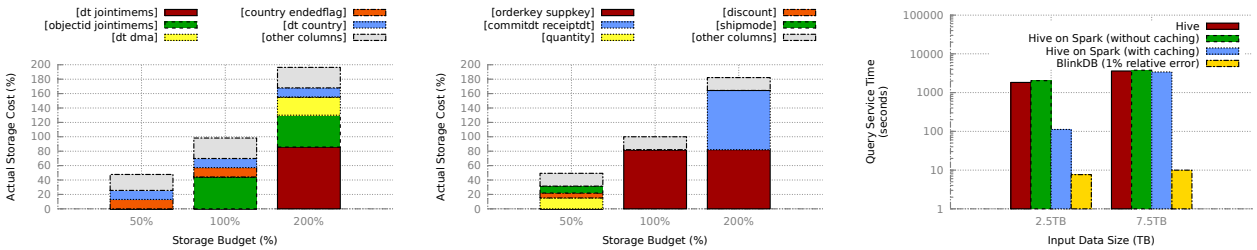
In this section, we evaluate BlinkDB's performance on a 100 node EC2 cluster using a workload from Conviva Inc. and the well-known TPC-H benchmark [3]. First, we compare BlinkDB to query execution on full-sized datasets to demonstrate how even a small trade-off in the accuracy of final answers can result 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. Third, we evaluate the effectiveness of our cost models and error projections at meeting the user's accuracy/response time requirements. Finally, we demonstrate BlinkDB's ability to scale gracefully with increasing cluster size.

7.1 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. We repeated the experiments with different storage budgets for the stratified samples– 50%, 100%, and 200%. A storage budget of x% in-



(a) Biased Samples (Conviva) (b) Biased Samples (TPC-H) (c) BlinkDB Vs. No Sampling

Figure 8. 8(a) and 8(b) 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. 8(c) compares the 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.

dicates that the cumulative size of all the samples will not exceed $\frac{x}{100}$ 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. 8(a) 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. 8(b) 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 paper are done with a 50% additional storage budget (*i.e.,* samples could use additional storage of up to 50% of the original data size).

7.2 BlinkDB 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 [22] on Hadoop MapReduce [2], Hive on Spark (called Shark [13]) – 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 8(c). Note that the Y axis is log scale. In all cases, BlinkDB significantly outperforms its counterparts (by a factor of 10 – 200×), 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 spilled to disk and the overall query response time is considerably longer.

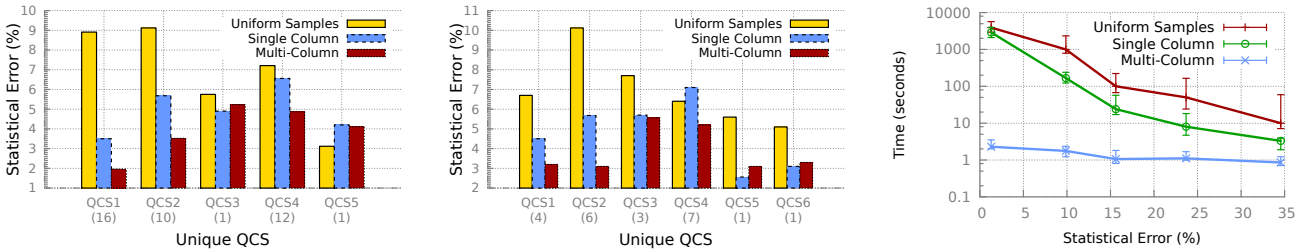
7.3 Multi-Dimensional Stratified Sampling

Next, we ran a set of experiments to evaluate the error (§7.3.1) and convergence (§7.3.2) 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 1 column.
3. **Multi-Dimensional Stratified Samples**. The sets of columns to stratify on were chosen using BlinkDB's optimization framework (§4.2), 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).

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



(a) Error Comparison (Conviva) (b) Error Comparison (TPC-H) (c) Error Convergence (Conviva)

Figure 9. 9(a) and 9(b) compare the average statistical error per QCS when running a query with fixed time budget of 10 seconds for various sets of samples. 9(c) compares the rates of error convergence with respect to time for various sets of samples.

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 9(a), and 9(b), 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.

7.3.2 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 §7.3, 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 9(c)) show that error bars from running queries over multi-dimensional samples converge orders-of-magnitude faster than random sampling (*i.e.,* Hadoop Online [11, 19]), and are significantly faster to converge than single-dimensional stratified samples.

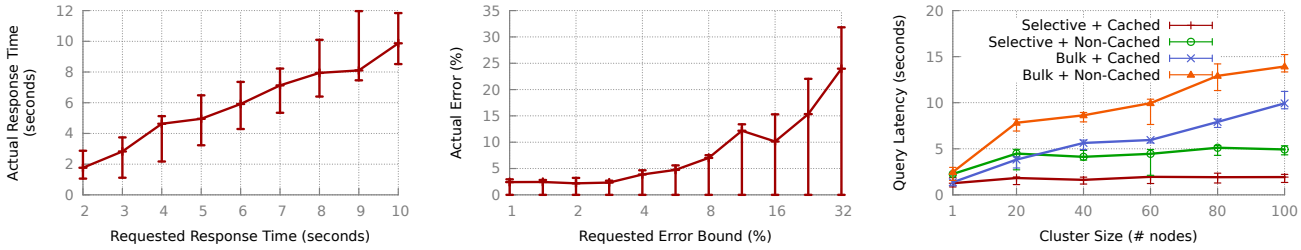
7.4 Time/Accuracy Guarantees

In this set of experiments, 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 10(a) 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 10(b) 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.

7.5 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 10(c) 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



(a) Response Time Bounds (b) Relative Error Bounds (c) Scaleup

Figure 10. 10(a) and 10(b) plot the actual vs. requested maximum response time and error bounds in BlinkDB. 10(c) plots the query latency across 2 different query workloads (with cached and non-cached samples) as a function of cluster size

memory these results indicate the min/max latency bounds for any query.

8. Related Work

Prior work on interactive parallel query processing frameworks has broadly relied on two different sets of ideas.

One set of related work has focused on using additional resources (*i.e.,* memory or CPU) to decrease query processing time. Examples include *Spark* [25], *Dremel* [17] and *Shark* [13]. While 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. By using samples, BlinkDB is able to scale better as the quantity of data grows. Additionally, being built on Spark, BlinkDB is able to effectively leverage the benefits provided by these systems while using limited resources.

Another line of work has focused on providing approximate answers with low latency, particularly in database systems. Approximate Query Processing (AQP) for decision support in relational databases has been the subject of extensive research, and can either use samples, or other non-sampling based approaches, which we describe below.

Sampling Approaches. There has been substantial work on using sampling to provide approximate responses, including work on stratified sampling techniques similar to ours (see [14] for an overview). Especially relevant are:

1. *STRAT* [10] builds a single stratified sample, while BlinkDB employs different biased samples. However, the more fundamental difference is in the assumptions and goals of the two systems. STRAT tries to minimize the expected relative error of the queries, for 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 query. Unfortunately, in many domains including those discussed in this paper, this assumption does not hold, since even queries with slightly different constants can have different FRs and thus, having seen one of them does not imply that STRAT can minimize the error for the other. In contrast, BlinkDB relies on the weaker assumption that the set of columns that have co-appeared in the past are likely to co-appear in the future too. Thus, instead of directly minimizing the error (which would be impossible without assuming perfect knowledge of future queries), BlinkDB focuses on maximizing the coverage of those column-sets, which as shown in §2, is much more suitable to ad-hoc workloads.

2. *SciBORQ* [21] 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. In contrast to BlinkDB, SciBORQ only supports time-based constraints. SciBORQ also does not provide any guarantees on the error margin.

3. *Babcock et al.* [9] also describe a stratified sampling technique where biased samples are built on a single column, in contrast to our multi-column approach. 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. Instead, BlinkDB runs on a single sample, chosen based on the current query.

4. *AQUA* [4, 6] 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, which can be prohibitive for large values of g (*e.g.,* in our workloads g exceeds 10). In contrast, BlinkDB considers only a small subset of these combinations by taking the data distribution and the past QCSs into account, at the expense of a higher storage overhead. In addition, AQUA always operates on the full sample, limiting the user's ability to specify a time or an error bound for a query. BlinkDB supports such bounds by maintaining multiple samples and employing a run-time sample selection module to select the appropriate sample type and size to meet a given query time or error bound.

5. *Olston et al.* [18] use sampling for interactive data analysis. However, their approach requires building a new sample

for each query template, while BlinkDB shares stratified samples across column-sets. This both reduces our storage overhead, and allows us to effectively answer queries for which templates are not known *a priori*.

Online Aggregation. Online Aggregation (OLA) [15] and its successors [11, 19] 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. However, in principle, techniques like online aggregation could be added to BlinkDB, to make it continuously refine the values of aggregates; such techniques are largely orthogonal to our ideas of optimally selecting pre-computed, stratified samples.

Materialized Views, Data Cubes, Wavelets, Synopses, Sketches, Histograms. There has been a great deal of work on "synopses" (e.g., wavelets, histograms, sketches, etc.) and lossless summaries (e.g. materialized views, data cubes). In general, these techniques are tightly tied to specific classes of queries. For instance, Vitter and Wang [24] use Haar wavelets to encode a data cube without reading the least significant bits of SUM/COUNT aggregates in a *flat* query[5], 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[6] when future queries are known in advance (modulo constants or other minor details). Nonetheless, these techniques are orthogonal to BlinkDB, as one could use different wavelets and synopses for common queries and resort to stratified sampling when faced with ad-hoc queries that cannot be supported by the current set of synopses. For instance, the join-synopsis [5] can be incorporated into BlinkDB whereby any join query involving multiple tables would be conceptually rewritten as a query on a single join synopsis relation. Thus, implementing such synopsis alongside the current set of stratified samples in BlinkDB may improve the performance for certain cases. Incorporating the storage requirement of such synopses into our optimization formulation makes an interesting line of future work.

9. Conclusion

In this paper, we presented BlinkDB, a parallel, sampling-based approximate query engine that provides support for ad-hoc queries with error and response time constraints. BlinkDB is based on two key ideas: (i) a multi-dimensional sampling strategy that builds and maintains a variety of samples, and (ii) a run-time dynamic sample selection strategy that uses parts of a sample to estimate query selectivity and chooses the best samples for satisfying query constraints. Evaluation results on real data sets and on deployments of up to 100 nodes demonstrate the effectiveness of BlinkDB at handling a variety of queries with diverse error and time constraints, allowing us to answer a range of queries within 2 seconds on 17 TB of data with 90-98% accuracy.

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

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

Acknowledgements

We are indebted to Surajit Chaudhuri, Michael Franklin, Phil Gibbons, Joe Hellerstein, our shepherd Kim Keeton, and members of the UC Berkeley AMP Lab for their invaluable feedback and suggestions that greatly improved this work. This research is supported in part by NSF CISE Expeditions award CCF-1139158, the DARPA XData Award FA8750-12-2-0331, and gifts from Qualcomm, Amazon Web Services, Google, SAP, Blue Goji, Cisco, Clearstory Data, Cloudera, Ericsson, Facebook, General Electric, Hortonworks, Huawei, Intel, Microsoft, NetApp, Oracle, Quanta, Samsung, Splunk, VMware and Yahoo!.

References

[1] Apache Hadoop Distributed File System. http://hadoop.apache.org/hdfs/.
[2] Apache Hadoop Mapreduce Project. http://hadoop.apache.org/mapreduce/.
[3] TPC-H Query Processing Benchmarks. http://www.tpc.org/tpch/.
[4] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *ACM SIGMOD*, May 2000.
[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *ACM SIGMOD*, June 1999.
[6] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. The Aqua approximate query answering system. *ACM SIGMOD Record*, 28(2), 1999.
[7] S. Agarwal, S. Kandula, N. Bruno, M.-C. Wu, I. Stoica, and J. Zhou. Re-optimizing Data Parallel Computing. In *NSDI*, 2012.
[8] G. Ananthanarayanan, S. Kandula, A. G. Greenberg, I. Stoica, and y. . . p. . . e. . h. b. . D. others title = Reining in the Outliers in Map-Reduce Clusters using Mantri, booktitle = OSDI.
[9] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *VLDB*, 2003.
[10] S. Chaudhuri, G. Das, and V. Narasayya. Optimized stratified sampling for approximate query processing. *TODS*, 2007.
[11] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears. Mapreduce online. In *NSDI*, 2010.
[12] G. Cormode. Sketch techniques for massive data. In *Synposes for Massive Data: Samples, Histograms, Wavelets and Sketches*. 2011.
[13] C. Engle, A. Lupher, R. Xin, M. Zaharia, et al. Shark: Fast Data Analysis Using Coarse-grained Distributed Memory. In *SIGMOD*, 2012.
[14] M. Garofalakis and P. Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, 2001. Tutorial.
[15] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.
[16] S. Lohr. *Sampling: design and analysis*. Thomson, 2009.
[17] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: interactive analysis of web-scale datasets. *Commun. ACM*, 54:114–123, June 2011.
[18] C. Olston, E. Bortnikov, K. Elmeleegy, F. Junqueira, and B. Reed. Interactive analysis of web-scale data. In *CIDR*, 2009.
[19] N. Pansare, V. R. Borkar, C. Jermaine, and T. Condie. Online Aggregation for Large MapReduce Jobs. *PVLDB*, 4(11):1135–1145, 2011.
[20] C. Sapia. Promise: Predicting query behavior to enable predictive caching strategies for olap systems. DaWaK, pages 224–233. Springer-Verlag, 2000.
[21] L. Sidirourgos, M. L. Kersten, and P. A. Boncz. SciBORQ: Scientific data management with Bounds On Runtime and Quality. In *CIDR'11*, 2011.
[22] A. Thusoo, J. S. Sarma, N. Jain, et al. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.
[23] S. Tirthapura and D. Woodruff. Optimal random sampling from distributed streams revisited. *Distributed Computing*, pages 283–297, 2011.
[24] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. SIGMOD, 1999.
[25] M. Zaharia et al. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012.
[26] M. Zaharia, A. Konwinski, A. D. Joseph, et al. Improving MapReduce Performance in Heterogeneous Environments. In *OSDI*, 2008.