Quickr: Lazily Approximating Complex AdHoc Queries in BigData Clusters

Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma,
Robert Grandl, Surajit Chaudhuri, Bolin Ding
Microsoft
msr-quickr@microsoft.com

ABSTRACT

We present a system that approximates the answer to complex adhoc queries in big-data clusters by injecting samplers on-the-fly and without requiring pre-existing samples. Improvements can be substantial when big-data queries take multiple passes over data and when samplers execute early in the query plan. We present a new *universe* sampler which is able to sample multiple join inputs. By incorporating samplers natively into a cost-based query optimizer, we automatically generate plans with appropriate samplers at appropriate locations. We devise an accuracy analysis method using which we ensure that query plans with samplers will not miss groups and that aggregate values are within a small ratio of their true value. An implementation on a cluster with tens of thousands of machines shows that queries in the TPC-DS benchmark use a median of 2× fewer resources. In contrast, approaches that construct input samples even when given 10× the size of the input to store samples improve only 22% of the queries, i.e., a median speed up of 0×.

1. INTRODUCTION

This paper considers the problem of approximating jobs in big-data clusters. Jobs specified as a mash-up of relational expressions and user-defined code increasingly dominate the big-data ecosystem, due in large part to the growth of frameworks such as Hive [40], Pig [37], SCOPE [18], Spark-SQL [12] and Dremel [33].

Queries in big-data clusters are approximatable but are complex and use many datasets. As an example, consider a production cluster at Microsoft with tens of thousands of machines that supports millions of queries per day for Bing and other services. (1) The distribution of queries over inputs is heavy-tailed. We find that the smallest subset of inputs which supports half of the queries is 20PB in size. The next 30% of queries touch another 40PB of inputs. (2) Further, queries are complex. They have many joins; median value is 3 and 90th percentile is 11. The execution graphs are deep, with the median graph having 192 operators and a depth of 28. Further, queries touch many columns from each dataset; the median query uses 8 columns per dataset and 49 at the 90th percentile. However, queries also have aggregations and their output is much smaller than the input indicating the potential for speed-up from approximation.

Approximating big-data queries, i.e., trading off degradation in answer quality to improve performance, has many use cases. We use performance colloquially here to refer to both query response time and cluster throughput. Since queries process large volumes of data on expensive clusters, even a modest decrease in resource usage, say 2×, would reduce the bill from Azure by 2× and production clusters which are capacity limited can run 2× more queries. Data scientists can tolerate imprecise answers for exploratory analysis and faster responses increase productivity [17, 25]. Two use cases are especially important: (a) queries that generate aggregated dashboard reports by analyzing logs if sped up would increase the refresh rate of the dashboards at no extra cost and (b) machine learning queries that build models by iterating over datasets (e.g., k-means) can tolerate approximations in their early iterations.

Unfortunately, state-of-art techniques cannot approximate complex queries. Most SQL databases and big-data systems offer the uniform sample operator. The user can sample as desired. But the systems do not reason about the accuracy of the resulting answer. A rich vein of prior research [6, 7, 9, 14, 20, 39] builds samples over input datasets. They deliver immense benefit to *predictable* queries that touch only one large dataset, i.e., any joins have to be with small dimension tables on foreign keys. However, they cannot support joins over more than one large table, queries that touch less frequently used datasets or query sets that use a diverse set of columns. As explained above, such queries and datasets dominate in big-data clusters. For example, on the TPC-DS [4] benchmark, our experiments show that when given 1×(4×) the size of the input to store samples, a state-of-the-art input sampling system BlinkDB [9] offers a benefit for 11% (17%) of the queries.

Our system, Quickr, has three goals. First, offer turn-key support for approximations: that is, given a query, decide whether or not it can be sampled and output an appropriate query plan with samplers. In particular, input samples, views or indices may not be available. Next, support complex queries, i.e., support the large portion of SQL shown in Table 1. Finally, ensure that answers will be accurate; that is, with high probability (whp), none of the groups will be missed in the answer and that the computed value of aggregations is within a bounded ratio of their true value (say ±10%). We are unaware of any system that achieves these goals.

A key observation in Quickr is that big-data queries perform multiple passes over data. This is partly due to the queries being complex and partly due to parallel plans. For example, a hash join requires two passes over data and one shuffle across the network. If data were sampled in the first pass, all subsequent computation could be sped up. The novel advantage of such *inline sampling* is that the gains from sampling have negligible apriori overhead. The disadvantage is that the *best case gains* are potentially smaller relative to apriori sampling which can execute the query on a very small

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA

ISBN 978-1-4503-3531-7/16/06…$15.00

DOI: http://dx.doi.org/10.1145/2882903.2882940

sample of input. In our cluster, the median query has 2.4 *effective*[1] passes over data and 6.5 passes at the 90th percentile. Furthermore, the first pass over data is often embarrassingly parallel. Hence, inline sampling offers sizable performance gains without apriori overhead.

We note that Quickr is complementary to building apriori samples. The latter is suited for simple predictable queries on popular datasets whereas Quickr can approximate the more complex queries over infrequently used datasets with zero overhead.

A novel aspect that lets Quickr cover many more queries than prior methods is a *universe sampler* that samples multiple join inputs. It is well known that joining a p probability sample of inputs is akin to a p^2 probability sample of the join output [21, 24]. Hence, sampling the join inputs improves performance at the cost of substantial degradation in answer quality. On the other hand, when queries join large inputs, sampling the join output offers limited speed-up. With the universe sampler, joining a p probability sample of inputs is statistically equivalent to a p probability universe sample of the join output. The key idea is to *consistently* sample the join inputs without any coordination (such as exchanging histograms of join keys [6, 22]). We will show that the universe sampler is applicable broadly, i.e., it supports multiple equi-joins and only requires the group-by columns and the value of aggregates to be uncorrelated with the sampled column set.

Quickr offers turn-key support for approximations by picking for every newly arriving query an execution plan with samplers. In this paper, we pick the best performing plan among those that have no worse than the specified amount of error. Since queries can be unapproximable (more on this later), a legitimate outcome is to output a plan without samplers. Quickr solves several challenges along the way. Which sampler to pick and where to place the samplers in the plan is a complex decision, in part, because Quickr uses three different samplers each of which is most appropriate under some conditions. The *universe* sampler is described above. The *uniform* sampler mimics a Bernoulli process. The *distinct* sampler ensures that no groups will be missed by *stratifying*, i.e., picking some rows for every distinct value of the group-by columns. To place these samplers at arbitrary locations in the plan, Quickr requires that samplers run in one pass over data, with bounded memory and in parallel; that is, different sampler instances independently process partitions of data.

There are two broad ways to find a good plan with samplers: add samplers on top of the plan output by a query optimizer or treat samplers as first-class operators within the query optimizer. Quickr adopts the latter approach because the former is computationally expensive and leads to poor quality plans. Since any operator can be followed by a sampler, there are exponentially many sampled plans. Reasoning about the performance and accuracy of each of these possible plans is complex and requires statistics that are harder to acquire outside the QO. Most importantly, adding samplers can lead to dramatically different query plans. For example, because a sampler reduces cardinality, joins may become implementable more cheaply or parallel portions of the plan can be replaced with the much simpler sequential plans. By treating samplers as first-class operators in a Cascades-style [27] cost-based query optimizer, Quickr effectively navigates a large space of possible sampled plans.

We call the resulting algorithm `ASALQA`, short for place **a**ppropriate **s**amplers at **a**ppropriate **l**ocations in the **q**uery plan **a**utomatically. `ASALQA` begins by placing a sampler optimistically before every aggregation. Then, several new transformation rules generate alternative plans by moving samplers closer to the input, i.e., to before database operators such as join, select, and project. `ASALQA` picks the best plan by costing both performance and accuracy. To do so, `ASALQA` uses statistics such as cardinality and distinct value counts which are computed in one effective pass by the first query that touches an input and are derived by `ASALQA` for sub-expressions. The plans output by `ASALQA` often have multiple samplers. Many plans have samplers deep in the query plan. `ASALQA` declares roughly 25% of the TPC-DS queries to be unapproximable for various reasons. An illustrative example is in §2.

Selection	Arbitrary (user-defined) expressions specified as $\mathbf{f}(col1, col2, \ldots)$ <=> $Constant$. Also, composing several such literals with ∨ or ∧.
Aggregates	DISTINCT, COUNT, SUM, AVG and their *IF equivalents. User-defined aggregates need annotations.
Join	All but full-outer join. Includes joins over multiple tables, outer joins and star queries with foreign key joins.
Others	Projects, Order By, Windowed aggregates, …

Table 1: Types of parallel SQL queries that are handled by Quickr. In particular Quickr can deal with arbitrary depth queries.

`ASALQA` can reason about the accuracy of a sampled expression. Our method transforms a query expression with arbitrarily many samplers to an equivalent expression with one sampler at the root. In particular, we generalize prior work that only considered `SUM`-like aggregates [35] to the case where answers can have groups. We also generalize the method to a broader class of samplers that are not generalized-uniform-samplers; our universe and distinct samplers are not uniformly random. Furthermore, we compute the error metrics for `ASALQA` plans in one effective pass over data whereas in general error bounds require a self-join [35] or bootstrap [10, 44].

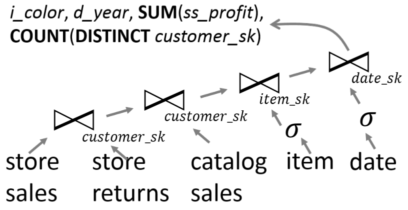
We have implemented Quickr in a production query optimizer. Experiments over queries in the TPC-DS benchmark show a median reduction of 2× in resources used; a few queries improve by over 5×. For over 90% of queries, Quickr does not miss groups. Most of the misses are due to `LIMIT 100` on the aggregation column. When considering the *full* answer, Quickr does not miss groups for 99% of queries. Aggregations are within ±10% of their true value in 80% of the queries; 92% of queries are within ±20%. We carefully explain the causes of high error. In contrast, BlinkDB [9] has a median gain of 0% even when given 10× the input size to store samples. That is, at least half the queries receive no benefit. In a parameter sweep, we find that the best coverage is 22% of queries and the best median speed-up *among the covered queries* is 35%.

To sum up, our key contributions are:

- Quickr offers a new way to lazily approximate complex adhoc queries with zero apriori overhead.
- Through careful analysis over queries in a big-data cluster, we find apriori samples are untenable because query sets make diverse use of columns and queries are spread across many datasets. The large number of passes over data per query makes the case for lazy approximations.
- We introduce a new sampler operator – universe – that effectively samples multiple join inputs.
- We consider query optimization over samples much more extensively. Our `ASALQA` algorithm automatically outputs sampled query plans only when appropriate.
- We present implementation results from a production big-data system.

We believe that this is just the first step towards practical lazy approximations. Queries can be sped up further by reusing sampled views [28] and by progressive execution [29]. In the rest of this paper, we describe our analysis of queries from our production cluster in §3. We describe our samplers in §4.1, the `ASALQA` algorithm in §4.2 and our accuracy analysis is in §4.3. Experimental results and a summary of related work are in §5 and §6.

[1] computed as $\left(\sum_{\text{task } t} \text{input}_t + \text{output}_t\right)$ / (job input + job output)



Technique	How sampled
Input sampling	Stratify store_sales on {item_sk, date_sk, customer_sk}
Quickr	Universe sample all three fact tables on customer_sk

Figure 1: A simple example that mimics many TPC-DS queries.

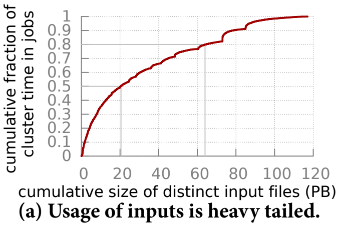
2. MOTIVATING EXAMPLE

Consider the query in Figure 1. Per item color and year, the query computes the total profit from store sales and the number of unique customers who have purchased and returned from stores and purchased from catalog. Item and date are dimension tables joined on a foreign key. The other three are large fact tables joined on shared keys. Since joining a pair of fact tables requires two reads and one shuffle, this query incurs many passes over its input data.

Stratification on a column set C guarantees that the sampler will pick some rows for every distinct value of C. The most useful input sample stratifies store_sales on {item_sk, date_sk, customer_sk} because store_sales is the largest table and not stratifying can miss groups in the answer. However, such a sample is as large as the input since the three columns have many distinct combinations leading to zero performance gains.

The universe sample on a column set C picks all rows whose hash of the value of columns in C falls in a randomly chosen subspace. Quickr yields a very different sampled query plan: at extraction on all fact tables, universe sample on customer_sk. This works because Quickr reasons that (a) universe sampling all fact tables on the join key customer_sk and then joining them is identical to a universe sample of the join output, (b) the group columns {i_color, d_year} are independent with customer_sk since different random subsets of customers have similar likelihood of purchasing different colored items over years and (c) the group has only a few distinct values and since the fact tables have over 10^9 rows, there are many rows per group. Interestingly, the universe sample can also estimate the number of unique customers, which is the same column whose value it sub-samples on. The reason is that the number of unique customers in the chosen subspace can be projected up by the fraction of subspace that is chosen. Quickr speeds up this query substantially. The data in-flight after the first pass reduces by the sampled probability and there are up to 4 subsequent passes over this dataset.

Note that small changes in the query can lead to very different sampled plans. If some user-defined operator applies on store_sales, Quickr *could* place the universe sampler *after* that UDO. Indeed, we find in our evaluation that many queries have samplers in the middle of the query plan. Consider a more substantial change: if the query only had store_sales, i.e., no joins with the other fact tables, Quickr would prefer a uniform sample of store_sales. As we show later, the uniform sampler has smaller variance than the universe sampler. If the answer were also grouped on i_category and i_name from item, the group could then contain so many distinct values that Quickr would distinct sample the store_sales table on ss_item_sk. That is, Quickr stratifies on join keys to mimic stratification on columns from item. Finally, if the answer has one group per day, that is group on d_date instead of d_year, then Quickr may declare the query unapproximable since stratifying store_sales on {ss_item_sk, ss_date_sk} will not reduce rowcount. In our evaluation, we see that all three samplers are used often. Many plans use multiple samplers. Quickr finds that 25% of the examined TPC-DS queries are unapproximable.



(a) Usage of inputs is heavy tailed.

Metric	Percentile value				
	25th	50th	75th	90th	95th
# of Passes over Data	1.83	2.45	3.63	6.49	9.78
1/firstpass duration fract.	1.37	1.61	2.09	6.38	17.34
# operators	143	192	581	1103	1283
depth of operators	21	28	40	51	75
# Aggregation Ops.	2	3	9	37	112
# Joins	2	3	5	11	27
# user-defined aggs.	0	0	1	3	5
# user-defined functions	7	27	45	127	260
size of QCS+QVS	4	8	24	49	104

(b) Characterizing some $O(10^8)$ queries that ran in a production big-data cluster over a two month span.

Figure 2: Analysis of the queries and datasets used in a large cluster. Equivalent estimates for publicly available benchmarks are in Table 9.

A ninja data scientist can probably reason about this large space of sampled plans quickly and correctly. Early in the project, it took us about an hour to manually analyze each query. Quickr offers a turn-key solution. In practice, Quickr also generates slightly better plans since we found that human experts fail to fully explore the space of possible plans. Also, experts did not have access to automatic derivations of cardinality and the number of distinct values.

3. PRIMER: APPROXIMABILITY OF BIG-DATA QUERIES

A key departure in Quickr is to inject sampling operators into the query plan. Suppose query Q over input I has answer $Q(I)$. Apriori sampling techniques search for the *best* sample of input I' such that $Q(I') \approx Q(I)$. Instead, Quickr searches over the space of sampled query plans for the *best* plan Q' such that $Q'(I) \approx Q(I)$. In both cases, *best* can imply best performance subject to the accuracy needs or vice-versa. Clearly the approaches are complementary. Here, we document aspects of queries observed in a large production cluster that lead us to believe that Quickr is necessary.

We analyzed the production queries in Microsoft's Cosmos clusters over a two month period. The clusters have tens of thousands of servers. Overall, $O(10^8)$ queries were submitted by $O(10^4)$ unique developers. The query language is a mash-up of relational and user-defined operations [18]. The query set includes: ad-hoc queries written by developers and production jobs which cook new data (e.g., ETL), or mine logs to construct dashboard reports. Figure 2 summarizes our findings.

Heavy Tail over Inputs: If most queries touch a small number of inputs, then it may be worthwhile to store apriori samples for those inputs. Per Figure 2a however, we find that queries access many different inputs. The figure is generated as follows for a two week period: (1) Per input, compute the total cluster hours used by queries that read the input. (2) When a query has multiple inputs, apportion its cluster hours among the inputs proportional to input size. (3) Sort inputs in decreasing order of their cluster hours. (4) Traversing in that order, compute the cumulative size of the input and the cumulative cluster hours. We see that jobs that account for half the cluster-hours touch 20PBs of distinct files. The last 25% of queries, the *tail*, access another 60PBs of files.

To store differently stratified samples, apriori storage techniques typically use sample storage of 1× to 10× the size of the input (eg. BlinkDB [9]). We see from Figure 2a that the smallest input set used by 20% of queries is 3PB. Hence, assuming input popularity can be predicted perfectly, covering 20% of queries will require between 3PB and 30PB of apriori samples. Such a large sample set is already a substantial fraction of the total input size (120PB). Quickr targets the vast majority of other queries and needs no apriori samples.

Recall that no stored samples implies that Quickr has to read all input data once. Per Figure 2b, the median query in the cluster takes 2.25 effective passes over data.[2] By sampling on the first pass, we can estimate that Quickr may speed-up the median job by 2.25×. 10% of queries can speed up by over 6×. The practical gains can be less because not all queries are approximable (for reasons below) or much more because the samplers can also speed up the computation on the first pass (e.g., fewer data to partition and write out).

Query Approximability: Query answers can be aggregates such as a `SUM`, `COUNT`, `AVG` or aggregations over groups such as `SELECT X, SUM(Y)`. The goal of approximation is to avoid processing all the data, yet still obtain (a) an unbiased estimate of the answer per group[3] that is within a small ratio of the true answer with high probability (or whp.), (b) ensure that no groups are missed whp., and (c) offer an estimate of the expected error per aggregation. Queries for which this is possible, we say, are *approximable*.

A key intuition behind the approximability of a query is the *support* per group of the answer. By *support* we refer to the number of data rows in the input that contribute to each group. Simple aggregations have support equal to the number of rows in the input. The support may vary across groups. In general, queries with large support receive high performance gains from approximation since even a small probability sample can produce an answer that is close to the true answer. If the underlying data value has high variance, more support is needed (more on this later). We observe that typical big data queries have large support due in part to their large inputs.

Per Figure 2b, typical queries also have joins, selects, projects and user-defined code, all of which further complicate approximability. We distinguish between row-local operations that take one or more columns and yield a column such as `DayOf(X : date)` and aggregations that take a collection of rows having a common value of the group-by columns and yield zero or more rows such as `X, MODE(Y)`. We call the former user-defined functions (UDFs) and the latter user-defined aggregates (UDAs). The median query has tens of UDFs and a few UDAs. The median query also has a handful of joins, several of which are not foreign-key joins.

To quantify the complexity of these operations, we co-opt the phrase Query Column Set (QCS) from BlinkDB [9] to refer to the subset of the input columns that appear in the answer or impact which rows belong in the answer. For example, the QCS of `SELECT X, SUM(Y) WHERE Z > 30` is {`X`, `Z`}. The corresponding **q**uery **v**alue **s**et (QVS) is {`Y`}, i.e., the columns that appear in aggregates. We recursively replace newly generated columns that appear in the QCS or QVS with the columns that were used to generate that column. We see that the size of QCS ∪ QVS is over 8 for 50% of queries. Further, the size of the median QCS is also 8. Comparing with equivalent estimates for benchmarks in Tables 3 and 9, we see that queries in the production cluster tend to be more complex. BlinkDB [9] constructs stratified samples on the QCS sets. Observing that the value of columns in the QVS can have high skew, StratifiedSampling [20] stratifies on QCS ∪ QVS.

Armed with the above background, we posit that apriori sampling has poor query coverage even when given storage space that is many times the size of that dataset. First, the QCSets have many columns. Since the goal of stratification is to store some rows for every distinct value of the columns in QCS, the more columns in the QCS, the more the distinct values and larger the stratified sample for that QCS. Second, queries have very diverse QCSets. Queries with different QCS will not benefit from the same stratified sample. Roughly, the smaller the intersection between the two QCSets, the less useful the sample will be. In the extreme case, when the QCSets have no overlap, sharing the sample can have the same error profile as a random sample[4] or worse if the QCSets have correlated values. As a result, given a storage budget, apriori sampling techniques must choose very carefully which QCSets to stratify on to help the largest set of queries [9, 20]. By injecting samplers into the query graph, Quickr completely avoids this problem.

Further, many queries that appear unapproximable for input samples can be sped up by Quickr. Consider a query with a large QCS. A stratified sample on that QCS may be as large as the input. However, Quickr can place a sampler *after* selections or joins whose (complex) predicates contributed many of the columns in the QCS. If the selects are pushed down to the first parallel pass on data, the gains from Quickr will be substantial.

Handling Joins: None of the known approximation systems handle joins well. Note that join between a fact and a dimension table is effectively a select since the foreign key relationship ensures that exactly one row will match out of the dimension table. Most prior work samples only one of the join inputs; doing so does not speed up queries where both input relations require a lot of work. To understand the issue with sampling both inputs, consider a two table join where T_1 has 80 rows with `X = 1` and T_2 has 1 row with `X = 1`. The join is on column X. To ensure that each tuple appears in the output with the same probability, a 25% sample on the join output requires 50% samples of both inputs.[5] That is, the join inputs have to be sampled with a quadratically higher probability. Even so, sample-and-join has a higher variance. For example, the probability that rows with `X = 1` disappear after sample-and-join is 0.5 whereas with join-and-sample, it is 10^{-10}.[6] The problem arises because the join output depends on the *joint distribution* over the two inputs but such joint distribution does not exist apriori. Further, it is expensive to compute when these relations are intermediate content generated in the midst of executing a deep query. A better sampler would have over-sampled the rows with `X = 1` in T_2 since that value occurs more often in T_1. Such correlation-aware samplers exist [6, 7, 21] but they are cumbersome to implement in parallel (since they build statistics and indices on one join input and use that to sample the other input) and are somewhat ineffective because the sampler cannot be pushed down further on the inputs. Quickr's universe sampler uniquely samples both join inputs without any data exchange between the inputs at runtime, thereby allowing Quickr to speed-up many more queries than prior work.

In summary, our analysis reveals the following:

- Distribution of queries over input datasets is heavy-tailed. Individual queries use many columns and a diverse set of

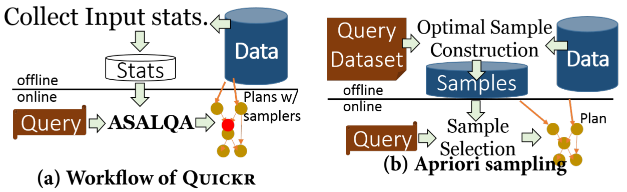
[2] computed as $\left(\sum_{\text{task } t} \text{input}_t + \text{output}_t\right) / (\text{job input} + \text{job output})$

[3] true sum ≈ 20 ∗ `sum` for a 5% sample. Also see Table 8.

[4] uniform sample = strat sampler with empty QCS.

[5] since a tuple will appear in output only if both its constituent tuples in the inputs are sampled and $0.25 = 0.5 * 0.5$

[6] T_1, T_2 and $T_1 \bowtie T_2$ have 80, 1, 80 rows with `X = 1` respectively. Hence, the probability of missing `X = 1` in join-and-sample is $(1 - 0.25)^{80} = 10^{-10}$ and in sample-and-join is $(1 - 0.5)^{80} + 0.5 = 0.5$.



(b) Apriori sampling

(a) Workflow of Quickr

Figure 3: Overview of Quickr and how it differs from prior methods.

columns such that the additional storage space required to store samples can be prohibitively large.

- Queries typically have aggregation operators, large support, and output ≪ input, so they are approximable.
- Several factors hinder approximability: queries use a diverse set of columns requiring extensive stratification. Many queries join large input relations.
- Queries are deep, involving multiple effective passes over data including network shuffles.

4. JUST-IN-TIME SAMPLING

Figure 3a shows an overview of Quickr. Quickr uses statistics of the input datasets to generate at query optimization time an execution plan with samplers placed at appropriate locations. The samplers are described in §4.1. The algorithm that determines how best to place the samplers is in §4.2. Analysis of the error and properties of the transformation rules is in §4.3. We briefly recount our goals:

- Minimal overhead to the administrator: That is, assume no apriori samples, indices or views and support ad-hoc queries.
- Support a large fraction of the queries in SQL and big-data scenarios; including general joins and UDFs.
- Performance gains should be sizable; either reducing the resource needs of a query or a faster completion time or both.
- Offer accurate answers: That is, with high probability miss no groups, offer confidence intervals, and estimate aggregate values to within a small ratio of their true values.

4.1 Samplers

Quickr uses three types of samplers. Each sampler passes a subset of the input rows. The subset is chosen based on the policies that we describe next. In addition, each sampler appends a metadata column representing the *weight* associated with the row. The weight is used to estimate the true value of aggregates and the confidence intervals. Our samplers are required to run in a *streaming* and *partitionable* mode. They have to execute in one pass over data with a memory footprint well below the size of the input or output. Furthermore, when many instances of a sampler run in parallel on different partitions of the input, the union of their output should mimic the output of one sampler instance examining all of the input. These minimal assumptions enable placing the samplers at arbitrary locations in a parallel query plan.

4.1.1 Uniform sampler

Given probability p, the uniform sampler Γ^{U}_{p} lets a row pass through with probability p uniformly-at-random. The weight column is set to $1/p$. In contrast, alternatives that pick a desired *number* of input rows uniformly-at-random with or without replacement [24] are neither streaming nor partitionable. If implemented with reservoir sampling so as to finish in one pass over data, their memory usage grows up to the desired output size and the parallel instances have to be synchronized and coordinated. The number of rows output by Γ^{U}_{p} is governed by a binomial distribution and each row can be picked at most once.

4.1.2 Distinct sampler

The uniform sampler is simple but it has some issues that limit it from being used widely. Queries with group-by such as `SELECT X, SUM(Y) GROUP BY X` can miss groups in the answer, especially those corresponding to values of `X` that have low support. For such queries, Quickr uses a distinct sampler which intuitively guarantees that at least a certain number of rows pass per distinct combination of values of a given column set. The distinct sampler also helps when aggregates have high skew. To see this problem, consider a three row input with the values 1, 1, 100 for column `Y`. The true answer for `SUM(Y)` is 102 but the projected answer changes dramatically based on whether the value of 100 is sampled or not; even at 50% sampling, the most likely answers are 2 and 202, each of which happen with likelihood 1/4.

Given a column set $\mathcal{C}$, a number δ, and probability p, the distinct sampler $\Gamma^{\mathrm{D}}_{p,\mathcal{C},\delta}$ passes at least δ rows for every distinct combination of values of the columns in $\mathcal{C}$.[7] Subsequent rows with the same value are let through with probability p uniformly-at-random. The weight of each passed row is set correspondingly; i.e., 1 if the row passes because of the frequency check and $1/p$ if it passes due to the probability check. Quickr picks the parameters $\{\mathcal{C}, \delta, p\}$ as a by-product of query optimization+sampling (§4.2)

To see how the distinct sampler improves over the uniform sampler, consider the following examples. Columns that form the group and those used in predicates can be added to the column set $\mathcal{C}$. Since the distinct sampler will pass some rows for every distinct value of the columns in $\mathcal{C}$, none of the groups will be missed and some rows will pass the predicate. Quickr also allows stratifying on functions over columns. For the skewed aggregates example (input has $\mathtt{Y} = \{1, 1, 100\}$) stratifying on $\lceil Y/100 \rceil$ ensures that $Y = 100$ will appear in the sample.

Since Quickr may employ the distinct sampler on any intermediate relation, the sampler must execute in a single pass, have a bounded resource footprint, and be partitionable. A naive implementation would maintain the observed frequency count per distinct value of column set $\mathcal{C}$. Then, it would pass a row while the frequency seen thus far is below δ with weight 1 and pick subsequent rows with probability p and hence a weight of $1/p$. This naive approach has three problems. The first problem is bias. The first few rows *always* pass through and are more likely to impact the answer. Worse, the first few rows *picked in the probabilistic mode* have a relatively disproportionate impact on the answer since their weight $1/p$ is much larger than the previous rows whose weight is 1. Only the more frequently occurring values of $\mathcal{C}$ are free from bias since enough rows will be picked for those values in the probabilistic mode. Second, the memory footprint can be as large as the number of distinct values in $\mathcal{C}$. Finally, when running in a partitioned mode, it is not possible to track how many rows with a particular value of $\mathcal{C}$ have been selected by the other (parallel) instances of the sampler. Hence, it is hard to ensure that all instances cumulatively pass *at least* δ rows and p *probability henceforth*.

Quickr solves the problems of the naive approach. To be **partitionable**, we carefully adjust δ based on the degree-of-parallelism of the sampler D. That is, each instance of the distinct sampler takes a modified parameter set $\{\mathcal{C}, \lceil \frac{\delta}{D} \rceil + \varepsilon, p\}$ wherein ε is carefully chosen to tradeoff between passing too many rows and passing too few rows by considering these two extreme cases–(1) all rows with the same value of $\mathcal{C}$ are seen by one sampler instance or (2) rows are uniformly spread across instances. The total number of rows passed by all instances is $(\delta/D) + \varepsilon$ for case (1) and $\delta + D\varepsilon$ for case (2). Case (1) is less frequent, but can happen if the input is ordered by the col-

[7] Precisely, at least $\min(\delta, \text{number of rows with that distinct value of } \mathcal{C})$

umn set $\mathcal{C}$. Quickr uses $\varepsilon = \delta/D$ since, in practice, the distribution of rows across instances more closely resembles case (2).

For **small memory footprint**, Quickr adapts a sketch that identifies heavy hitters in one pass over data [32]. Crucially, using this sketch Quickr maintains *approximate* frequency estimates for only the *heavy hitters* in memory that is *logarithmic* in the number of rows. Our key insight is that the distinct sampler's gains arise from probabilistically dropping rows that correspond to values of $\mathcal{C}$ that occur very frequently; tracking only the heavy-hitters achieves most of these gains. In particular, for an input of size N and constants s, τ, our sketch identifies values with frequency above $sN \pm \tau N$ and estimates their frequency to within $\pm\tau N$ off their true frequency. The memory usage is $\log(\tau N)/\tau$. Quickr uses $\tau = 10^{-4}$ and $s = 10^{-2}$ for a memory footprint of 20MB with $N = 10^{10}$ input rows.

To reduce **bias**, Quickr holds in a reservoir rows that are *early* in the probabilistic mode and passes them with the correct weight. In more detail: per distinct value, pass the first δ rows with weight 1. Subsequently, maintain a reservoir of size S. When more than $\delta + S/p$ rows are seen, flush the rows held in the reservoir with weight $1/p$. From then on, pick rows with probability p, i.e., without a reservoir. When the sampler has seen all rows, flush the rows in all non-empty reservoirs with weight $(\text{freq} - \delta)/\text{S}$ where freq is the number of observed rows. To see this method in action, suppose $\delta = 10, p = 0.1, S = 10$. It is easy to see that distinct values of $\mathcal{C}$ with freq in $[1, 10]$ will not use a reservoir. All their rows pass with weight 1. Those with freq in $(\delta, \delta + S/p] = [11, 110]$ are more interesting. For a value with freq of 30, its first ten rows pass right away, the next twenty go into the reservoir and a random subset of ten rows will be flushed *at the end* with a weight of $(\text{freq} - \delta)/\text{S} = 2$. Notice that the probability of a row numbered in $[11, 30]$ to be emitted is $1/2$. Finally, values with freq above 110 *lose* their reservoir once the 111'st row is seen. At that point, ten rows from the reservoir are passed with weight of 10. This method reduces bias because samples passed by the reservoir receive a weight that exactly corresponds to the probability with which the row was chosen. Further, only a small reservoir is kept (no more than S) and only for distinct values that have observed frequency between δ and $\delta + S/p$. Hence, the memory footprint is much smaller than straightforward reservoir sampling.

In summary, we are not aware of any other stratified sampler that functions in a streaming and partitionable manner. Furthermore, we believe that supporting stratification over functions of columns is novel. Though we use a column set $\mathcal{C}$ in the above description, this sampler can support a vector of functions whose domain is a subset of $\mathcal{C}$. This allows for *just enough* stratification.

4.1.3 Universe sampler

Universe sampler is a new operator that uniquely allows Quickr to sample the inputs of joins. Consider this example:

```
SELECT COUNT(DISTINCT order), SUM(ws.profit)
FROM ws JOIN wr ON ws.order = wr.order
```

Web-sales (ws) and web-returns (wr) are large fact tables being joined on a shared key. As discussed in §3, uniform sampling both the join inputs is not useful. Distinct sampling both the inputs has limited gains if the join keys have many columns and hence, many distinct values. *Correlation-aware* samplers [6, 21, 22, 24] are inefficient (since they construct histograms or indices on join inputs) and ineffective (since they require samplers to immediately precede the join). More details are in §6. Generalizing this example, similar cases happen with self-joins and *set operations* such as counting the number of orders that occur in one table, both tables, or exactly one table. All such cases are approximable (have aggregations and output ≪ input) but existing samplers do not help.

We now explain the insight behind universe sampler. Much of the trouble in sampling join inputs arises because the inputs have to *sampled independently* and their *joint behavior* has to be statistically meaningful. Suppose the value of the join keys is projected into some high dimensional *space* (e.g., using a hash function) and samplers on both inputs pick the ***same*** *random* portion of this space. That is, on both join inputs pick all rows whose value of join keys falls in some chosen subspace. For example, pick from the tables `ws` and `wr` rows that have `Hash(order)%4 = 2`. This yields a 25% sample. A join over these samples is statistically equivalent to sampling after the join (i.e., picking rows in join output that belong to the chosen subspace). Our implementation uses a cryptographically strong hash function. Hashing lets us pick a subspace without apriori knowledge of the *range* of values and we can vary the desired sample size by choosing a corresponding portion of the hash range.

More formally, the universe sampler takes as input a column set $\mathcal{C}$ and a fraction p. It chooses a p fraction of the value space of the columns in $\mathcal{C}$. And, passes all rows whose value of columns in $\mathcal{C}$ belong to the chosen subspace. Related pairs of samplers will pick the same subspace. Note that the universe sampler is partitionable and needs only one pass: whether or not a row passes depends only on the values of the columns in $\mathcal{C}$, so the sampler keeps no state across rows. As a corollary, different instances that process different subsets of the input would make the same decisions.

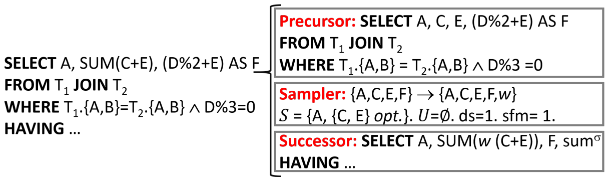
4.1.4 Limitations and interactions between samplers

We summarize the applicability of each sampler (e.g., guard conditions) and how the samplers complement each other and together expand the applicability of samplers as a family. All three samplers are broadly applicable in the sense that we can ensure commutativity with other database operations; although analyzing the accuracy of answers generated by universe and distinct samplers requires new methods because these samplers are not uniformly random (§4.3). Among the samplers, the uniform sampler is the most general. Quickr uses the uniform and universe samplers only when the stratification requirements, if any, can be met. That is, either there is high support per group or the columns that need stratification (e.g., group by columns) are independent with the universe columns. For some aggregations, such as `COUNT` and `COUNT DISTINCT`, we note that column independence is not needed for the answer to be unbiased (e.g., `COUNT DISTINCT order` in §4.1.3). The universe sampler is applicable for equi-joins over arbitrarily many columns. Further, the universe sampler can be used for multiple joins in a query as we saw in Figure 1. More precisely, universe sampling can be used for *exactly one set of columns* in any query sub-tree that has an aggregate at the root. The three samplers together expand the range of applicability of samplers. That is, by integrating with the query optimizer (§4.2), Quickr considers various join orders, choices of stratification and/ or universe columns, and choices of sampler locations to pick an appropriate plan. Consequently, our results will show that many otherwise unapproximable queries benefit from Quickr.

4.2 Samplers + QO (or ASALQA)

Since every operator in the plan can potentially be followed by a sampler, the search space of possible sampled plans is very large. Further, for each possible choice of samplers and locations in the plan, one has to reason about the *performance and accuracy* of the corresponding plan. Given an input query, `ASALQA` outputs an execution plan with appropriate samplers inserted at appropriate locations. Our current target is the plan that achieves the best performance with accuracy as the constraint.

There are at least two choices as to how we can obtain a good plan that contains samplers: (a) Insert samplers a posteriori into a



SELECT A, SUM(C+E), (D%2+E) AS F
FROM T_1 **JOIN** T_2
WHERE T_1.{A,B}=T_2.{A,B} ∧ D%3=0
HAVING …

Precursor: SELECT A, C, E, (D%2+E) AS F
FROM T_1 **JOIN** T_2
WHERE T_1.{A,B} = T_2.{A,B} ∧ D%3 =0

Sampler: {A,C,E,F} → {A,C,E,F,*w*}
S = {A, {C, E} *opt.*}. *U*=∅. ds=1. sfm= 1.

Successor: SELECT A, SUM(*w* (C+E)), F, sum$^{\sigma}$
HAVING …

Figure 4: Seeding samplers into the query.

plan that is output by a traditional relational query optimizer or (b) Incorporate samplers as first-class operators along with the other database operators and explore the larger combined space of possible plans within a query optimizer. Notice that option (b) can yield plans that cannot be obtained from using option (a). For example, when a sampler reduces cardinality downstream join operations can be implemented differently and more efficiently as a cross join instead of a hash-join [8]. As another example, for queries with many joins and selects, option (a) may offer a plan on which all simple edits to insert samplers appear infeasible (inaccurate). Yet, a different ordering of the joins or selects may allow samplers to be inserted. Hence, we chose option (b); we offer a new ASALQA algorithm that incorporates samplers as native operators into a Cascades-style cost-based optimizer [18, 27].

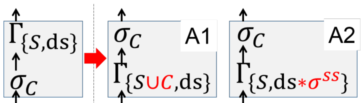
Query optimization in Cascades consists of two main phases. In the *logical plan exploration* phase, a set of transformation rules generate alternative plans. The *physical plan creation* phase converts each logical operation to a physical implementation. ASALQA modifies both these phases and proceeds as follows. Samplers are injected into the query execution tree before every aggregation (§4.2.2). Intuitively, this represents the *potential to approximate* at that location. Next, Quickr has a set of new transformation rules that push samplers closer to the raw input (§4.2.3–§4.2.5). The alternatives generated by a rule have no worse accuracy but can have better performance. Furthermore, a new rule changes the degree-of-parallelism (§A) of sampled query sub-expressions which can in turn trigger other changes to the overall plan. Finally, plan costing uses data statistics to identify the best plan, both in terms of performance and accuracy (§4.2.6, §4.3).

4.2.1 Sampler: logical and physical state

During the logical exploration phase, the requirements on a sampler are encoded in what we call the *logical state*. The requirements of the sampler change when the samplers are moved by the transformation rules. Furthermore, they may be implementable by one or more physical samplers. After logical exploration, ASALQA picks the best sampler that meets the requirements (§4.2.6). We denote the logical state by $\{S, U, \mathsf{ds}, \mathsf{sfm}\}$. S and U are the columns that the sampler needs to stratify or universe sample upon respectively. We also refer to them as *strat cols* and *univ cols* respectively. ds and sfm are short for downstream selectivity (i.e., the cumulative selectivity of operators between the sampler and the answer) and stratification frequency multiplier; their use will be described shortly.

4.2.2 Seeding samplers

We seed samplers by replacing each statement that has aggregations with three statements–a precursor, a sampler and a successor–as shown in Figure 4. This is optimistic; that is, ASALQA replaces the sampler with a pass-through operation if the error goal cannot be met. The *precursor* mimics the original statement but for aggregations. In particular, the precursor receives all of the JOIN clauses, WHERE clauses, UDOs and AS projections from the original statement. Aggregations in the precursor are replaced with their corresponding input columns. The *successor* performs these aggregations by replacing each with (a) an unbiased estimator of the true value computed over the sampled rows and (b) appends an optional column that offers a confidence interval. Table 8 shows how Quickr rewrites some example aggregations. The successor also receives the HAVING clause from the original statement. A *sampler* statement is introduced between the precursor and successor. Columns that appear in the answer, e.g., A in Figure 4, are added to the stratification column requirement (S) in the sampler. Further, columns in the *IF clauses, in SUM and in COUNT(DISTINCT) (e.g., C, E in Figure 4) are optionally added to the set S (details in §4.2.3). Recall from §4.1 that stratification on these columns ensures that no groups are missed and corrects for value skew. Figure 4 also shows the initial values for the rest of the logical state ($U = \emptyset; \mathsf{ds} = 1; \mathsf{sfm} = 1$).



$\Gamma_{\{S,\mathrm{ds}\}}$
σ_C
σ_C
A1
$\Gamma_{\{S \cup C,\mathrm{ds}\}}$
σ_C
A2
$\Gamma_{\{S,\mathrm{ds} * \sigma^{ss}\}}$

Figure 5: Pushing sampler past select: See §4.2.3.

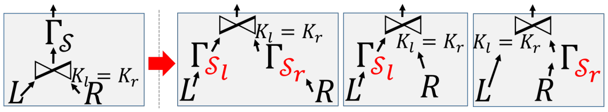


Figure 6: Pushing sampler past join: See §4.2.4.

4.2.3 Pushing sampler past Select

We start with a simple yet non-trivial transformation rule that pushes samplers past select operators. For the expression on the left in Figure 5, ASALQA generates the two alternatives on the right. Here σ_C denotes a select that uses columns C in its predicate. Let σ^{ss} be its selectivity.[8] We only show the relevant fields in the logical sampler state. To understand why this rule helps, note the following logic.

Alternative A1 (Figure 5 middle) stratifies additionally on the predicate columns C. Doing so guarantees that the error will be no worse than the expression on the left since at least some rows pass the sampler for every distinct value of C. Unfortunately, the performance can be worse by a lot. This is because the more columns in C, the greater the number of distinct values in the stratified column set $S \cup C$, forcing the sampler to pass many more rows. The second alternative A2 (Figure 5 right) retains perf gain potential at the cost of additional error. If the sampler were to not stratify on the columns in C, its performance is no worse than before. But, there will be fewer rows in the answer by a factor of σ^{ss}. Therefore, this plan is more likely to miss groups or have a higher variance for aggregates. It is easy to see when each alternative is preferable: (a) if the select has many predicate columns but is not very selective, A2 is better; (b) if the select is highly selective, A1 may be better; (c) otherwise, neither alternative is better.

Observe also that in both alternatives the sampler reduces the work for the select since it now operates on fewer rows. If the select is a conjunctive predicate ("and"), the logic above is applied per conjunction. When some predicate columns C are already in the strat cols S, observe that A1 needs no correction. For A2, ASALQA uses a heuristic to reduce ds by a smaller amount than σ^{ss}.

4.2.4 Pushing sampler past Join

We highlight the transformation rules to push sampler past an equi-join because it has deep implications on performance and illustrates novel aspects of our technical contributions. As shown in Figure 6, the sampler can be pushed down to one or both inputs. Let L, R be the input relations and K_l, K_r be the corresponding join keys. When pushing a sampler past join, two key considerations arise: (i) continue to meet the stratification and universe requirements in the

[8] σ^{ss} = rows in output of σ/rows in input.

```
Inputs:
      $\mathcal{S}$:    Sampler state = $\{S, U, \text{sfm}, \text{ds}\}$
   $L, R$:    relational inputs of join with columns $L^c, R^c$ respectively
 $K_l, K_r$:    Columns used as join keys // assume equi-join
Output: Vector of alternate samplers on one or both inputs of join.
```

```
1  Func: NumDV$(R, C)$ : // distinct value count of columns $C$ in $R$
```

```
2  Func: ProjectColSet$(S, \pi_{C_1 \to C_2})$: // $S, C_1, C_2$ are column sets.
   $|C_1| = |C_2|$. Replace columns in $S \cap C_1$ with corresponding ones in $C_2$.
```

```
3  Func: OneSideHelper$(\mathcal{S}, L, R, K_l, K_r, U_l)$:
4  $\vec{\mathcal{A}} \leftarrow \{\}$ // vector of state of alternative samplers on left relation
5  $S_f \leftarrow$ ProjectColSet$(\mathcal{S}.S, \pi_{K_r \to K_l})$//Normalize; these are “full” strat cols
6  $S_l \leftarrow S_f \cap L^c$// strat cols defined on left
7  sfm $\leftarrow \mathcal{S}$.sfm
8  if $|S_f - S_l| > 0$ and $|K_l - S_l| > 0$// missing some strat cols then
9      sfm $\leftarrow$ sfm $* \frac{\min(\text{NumDV}(L, K_l - S_l),\ \text{NumDV}(R, S_f - S_l))}{\text{NumDV}(R, \text{ProjectColSet}(K_l - S_l, \pi_{K_l \to K_r}))}$
10     $S_l \leftarrow S_l \cup K_l$ // append join keys to strat cols
11 $K_{\text{rem}} \leftarrow K_l - S_l$
12 foreach subset $S$ of $K_{rem}$ do
13     $S_{\text{skip}} \leftarrow K_{\text{rem}} - S$// will strat on $S$, these join keys remain.
14     ds $\leftarrow \mathcal{S}.ds$ / NumDV$(L, S_{\text{skip}})$ $*$ // penalize by increasing ds
         $\min\left(\text{NumDV}(L, S_{\text{skip}}),\ \text{NumDV}(R, \text{ProjectColSet}(S_{\text{skip}}, \pi_{K_l \to K_r}))\right)$
15     // omitted detail: check for dissonance
16     $\vec{\mathcal{A}} \leftarrow \{S_l \cup S,\ U_l,\ \text{sfm},\ \text{ds}\}$
17 return $\vec{\mathcal{A}}$
```

```
18 Func: PrepareUnivCol$(U, K)$:
19 if $U = \varnothing$ or $U = K$ then return $K$ // Else, cannot do universe ;
```

```
20 Func: PushSamplerOnOneSide$(\mathcal{S}, L, R, K_l, K_r)$:
21 $U_l \leftarrow$ ProjectColSet$(\mathcal{S}.U, \pi_{K_r \to K_l})$
22 if $U_l - L^c = \varnothing$// can push to one side iff other side has no univ col. then
23     return OneSideHelper$(\mathcal{S}, L, R, K_l, K_r, U_l)$
24 return $\{\}$ // no alternatives
```

```
25 Func: PushSamplerOntoBothSides$(\mathcal{S}, L, R, K_l, K_r)$:
26 $U_l \leftarrow$ PrepareUnivCol(ProjectColSet$(\mathcal{S}.U, \pi_{K_r \to K_l}), K_l)$
27 $U_r \leftarrow$ PrepareUnivCol(ProjectColSet$(\mathcal{S}.U, \pi_{K_l \to K_r}), K_r)$
28 if $U_l = \varnothing$ or$U_r = \varnothing$ then return $\{\}$// cannot use univ ;
29 $\vec{\mathcal{A}}_l \leftarrow$ OneSideHelper$(\mathcal{S}, L, R, K_l, K_r, U_l)$
30 $\vec{\mathcal{A}}_r \leftarrow$ OneSideHelper$(\mathcal{S}, R, L, K_r, K_l, U_r)$
31 return $\vec{\mathcal{A}}_l \times \vec{\mathcal{A}}_r$ // cross product, output is vector of state-pairs
```

Figure 7: Pseudocode for pushing samplers past join.

sampler's logical state $\mathcal{S}$ and (ii) account for the additional changes to the answer due to the join following the sampler.

The PushSamplerOnOneSide function in Figure 7 shows how ASALQA considers pushing the sampler to the left input. Its goal is to find sampler $\mathcal{S}_l$ such that $\Gamma_{\mathcal{S}_l}(L) \bowtie R$ is an alternate for $\Gamma_{\mathcal{S}}(L \bowtie R)$. We first try to satisfy the universe and stratification requirements in $\mathcal{S}$. If any of the universe columns $\mathcal{S}.U$ appear only on the right relation, then pushing to the left is not possible; since some univ cols are unavailable on the left, picking rows in the desired value subspace of $\mathcal{S}.U$ is not possible. (See the check for $U_l - L^c = \varnothing$ in PushSamplerOnOneSide.) If some of the strat cols $\mathcal{S}.S$ are missing on the left, however, ASALQA stratifies on the left join keys. (See lines 8–10 in OneSideHelper.) In the example in Figure 1, stratifying `store_sales` ⋈ `date` on `d_year` can be approximated by stratifying `store_sales` on the left join key `sold_date_sk`. Intuitively, stratifying on join key ensures that some rows will appear on the sampled left to match every row from the right. However, the join keys may have more or fewer distinct values than the columns that they replace. In the above example, the join key has 365× more distinct values than `d_year`; hence the support per group *appears* much smaller than it is. ASALQA uses sfm (stratification frequency multiplier) to correct for this difference. Intuitively, when replacing a stratified column with a join key having more (or fewer) distinct values, the value of sfm goes up (or down) and sfm is used as a multiplier when computing the group support (§4.2.6). We note that this has been a crucial enabler in pushing samplers onto large relations while properly accounting for stratification needs. Finally, accounting for the additional changes to the answer due to the join is similar to the case of select (see §4.2.3). Either the new sampler stratifies on the join keys or gets a smaller downstream selectivity. Lines 12–16 of OneSideHelper show how ASALQA considers different subsets of K_{rem} to add to strat cols and adjusts ds accordingly.

row count	
Average*, Variance* Number of Distinct Values Heavy-Hitter Values and Freq.	per *interesting*† column

* : for columns with numerical values; † : columns that appear in select filters, join clauses or contained in the eventual answer.

Table 2: Statistics used by QUICKR to facilitate sampler selection

The PushSamplerOntoBothSides function in Figure 7 shows how ASALQA considers pushing a sampler to both join inputs. Its goal is to find samplers $\mathcal{S}_l, \mathcal{S}_r$ such that $\Gamma_{\mathcal{S}_l}(L) \bowtie \Gamma_{\mathcal{S}_r}(R)$ is an alternate for $\Gamma_{\mathcal{S}}(L \bowtie R)$. Note the calls to OneSideHelper to push on to each side of the input. The only substantial change is adding join keys to the corresponding univ cols. As shown in PrepareUnivCol (line#19), ASALQA adds new universe requirements if univ cols do not exist already ($U = \varnothing$) or if they are identical to the join keys ($U = K$) as in the example query in Figure 1. As before, ASALQA picks the best option from among these alternatives.

ASALQA checks for dissonance between the stratification and universe requirements (line#15 in Figure 6). Columns that appear in both the S and U sets are troublesome because the universe sampler will only pick a subset of the values of such columns whereas stratification requires all values. ASALQA allows overlap in the requirements only when $|S \cap U| \ll |S|$; i.e., if only a few columns overlap, the column sets can be considered effectively independent. Further, overlap is allowed for columns that appear in S only because of `COUNT` or `COUNT DISTINCT` since such aggregates can be estimated correctly (see Table 8).

4.2.5 Other transformation rules, Parallel plans and Global constraints

We conclude the discussion of transformation rules by noting that ASALQA pushes samplers past many other operators including projections, union-alls and outer joins. In some cases, it is strictly better to push down the sampler. In other cases, ASALQA uses the costing that will be described in §4.2.6 to decide whether pushing down a sampler is better and to pick among the various choices. We mention two important issues here, the details of which are in §A. First, for the universe sampling property to hold, both input relations of a join should have an identical universe sampler (same column sets and probability). We ensure that this and other such *global requirements* are satisfied on the bottom-up pass of the query optimization. Next, parallel plan performance can improve further if samplers are followed by exchanges since the cardinality reduction due to the sampler can translate into a degree-of-parallelism reduction leading to more efficient serial sub-plans or better implementation choices. More details are in §A.

4.2.6 Costing Sampled Expressions

In this section, we describe how to cost sampled expressions. Costing helps pick between alternatives and determines how best to implement a sampler while meeting all of its requirements.

A key input to costing is the cardinality estimates per relational expression (how many rows) and the number of distinct values in each column subset. Table 2 shows the statistics that Quickr collects for each input table. If not already available, the statistics are computed by the *first* query that reads the table [8]. Using these input statistics, `ASALQA` *derives* the corresponding statistics for each query sub-expression. The derivation improves upon prior work [16] by using heavy hitter identity and frequency.

Armed with the above stats, we reason about how best to implement a sampler so as to meet the requirements encoded in its logical state $\mathcal{S} = \{S, U, \text{sfm}, \text{ds}\}$. We use two high level simplifications. To ensure that the performance gains are high, we disallow sampling with probability above 0.1. Next, we use a fixed error goal: with high probability, do not miss groups in answer and keep aggregate value within ±10% of the true value. We defer a more graceful trade-off between speed-up and accuracy to future work.

Meeting these goals translates to the following sequence of checks: (C1) Is stratified column requirement S empty or can some sample probability $p \in [0, 0.1]$ ensure that, with high probability, each distinct value of the columns in S receives at least k rows? (C2) Is univ col requirement U empty? Answering C1 requires the cardinality and distinct value derivations described above for the input relation. Further, these numbers are multiplied by $\mathsf{ds} * \mathsf{sfm}$. Recall that downstream selectivity ds is the probability that a row passed by this sampler will make its way to the answer (after downstream selections or joins). And, sfm is a multiplier to correct the effect of replacing strat cols with join keys (§4.2.4). If the answer to both C1 and C2 is true, the sampler is implemented using the uniform sampler (§4.1.1). If only C1 is true, a universe sampler is chosen. If only C2 is true, a distinct sampler may be chosen; we check whether there will be any data reduction, i.e., at least k_l rows exist per distinct value of columns in S. We choose $k_l = 3$. The default option is to not-sample i.e., implement sampler as a pass-through operator.

Intuitively, the reasoning is that if S and U are empty or if there is substantial support, a uniform sampler suffices. Universe sampler has higher variance and is chosen only when needed ($U \neq \varnothing$) and stratification needs are met ($S = \varnothing$, or enough support, or $|S \cap U| \ll |S|$). Finally, a query plan without samplers is chosen and is the desired option when (a) the per-group support is small or (b) downstream selectivity ds is so small or stratification requirements are so restrictive that the number of rows per distinct value of S is below k_l. Physical sampler parameters (e.g., sample probability p, δ for distinct) are chosen as the smallest values that satisfy (C1). We use $k = 30$ because anecdotally 30 samples are needed by central-limit-theorem which we use to estimate confidence intervals. In our evaluation, a parameter sweep shows that the plans output by `ASALQA` are similar for $k \in [5, 100]$.

4.3 Accuracy analysis

Given a query plan $\mathcal{E}$, with many samplers at arbitrary locations, Quickr offers unbiased estimators of aggregate values as well as the probability of missing groups and the confidence intervals for aggregates. We briefly describe how to do this. First, suppose that a sampler immediately precedes the aggregation and group by operator. We use the well-known *Horvitz-Thompson (HT) estimator* [24] to calculate unbiased estimators of the true aggregate values and the variance of these estimators. The details are in §B.1. Then, we can use central-limit-theorem to compute confidence intervals. This does not suffice, however, because the samplers in $\mathcal{E}$ can be arbitrarily far away from the aggregates. In fact, Quickr pushes samplers further away to improve performance.

Next, to analyze the case when samplers are at arbitrary locations we introduce a novel notion of *dominance* between query expressions whose output is identical when samplers are removed from both: $\mathcal{E}_2$ is said to dominate $\mathcal{E}_1$ denoted as $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$, iff the *accuracy* of $\mathcal{E}_2$ is no worse than that of $\mathcal{E}_1$. This definition mathematically distills the necessary and sufficient conditions to ensure that $\mathcal{E}_2$ has no worse variance of estimators and no higher probability of missing groups than $\mathcal{E}_1$. The details are in §B.2, Proof of Proposition 5.

We then show that dominance transitively holds across database operators. Suppose that $\pi, \sigma, \bowtie$ denote a project, select and join.

Proposition 1 (Dominance Transitivity)**.** *For pairs of expressions $\mathcal{E}_1$, $\mathcal{E}_2$ and $\mathcal{F}_1$, $\mathcal{F}_2$ that are equivalent if all samplers were removed:*

i) $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$ *implies* $\pi(\mathcal{E}_1) \overset{*}{\Rightarrow} \pi(\mathcal{E}_2)$*;*

ii) $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$ *implies* $\sigma(\mathcal{E}_1) \overset{*}{\Rightarrow} \sigma(\mathcal{E}_2)$*;*

iii) $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$ *and* $\mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{F}_2$ *implies* $\mathcal{E}_1 \bowtie \mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{E}_2 \bowtie \mathcal{F}_2$*, if samplers in $\mathcal{E}_i$ are independent on samplers in $\mathcal{F}_i$ or $\mathcal{E}_i$ and $\mathcal{F}_i$ share the same universe sampler.*

The proof is in §B.2 Proposition 1.

To analyze the accuracy of a plan, we use the dominance rules to inductively *unroll* `ASALQA`. That is, just for the sake of analysis, we find an equivalent query expression $\mathcal{E}'$ which has only one sampler just below the aggregation such that $\mathcal{E}' \overset{*}{\Rightarrow} \mathcal{E}$. We use the above HT estimators on $\mathcal{E}'$. By dominance, the accuracy of $\mathcal{E}$ is no worse. An illustration of this process for the query in Figure 1 is in Figure 9.

Finally, we can prove that the analysis above requires only one effective pass over the sample; the details are in §B.1 Proposition 2.

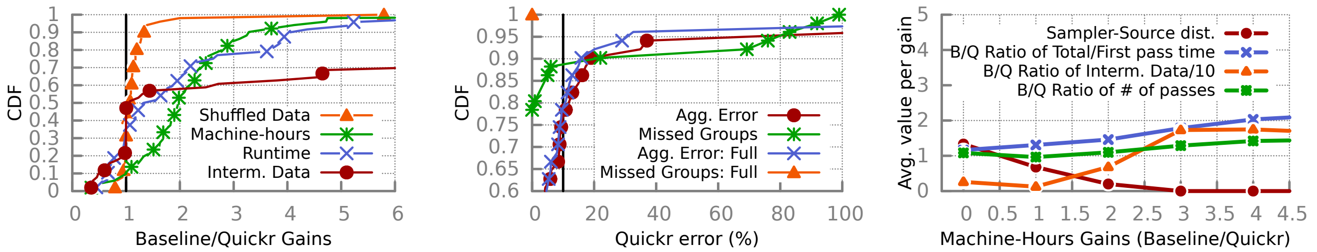
Proposition 2 (Complexity)**.** *For each group G in the query output, Quickr needs $O(|\mathcal{E}(G)|)$ time to compute the unbiased estimator of all aggregations and their estimated variance where $|\mathcal{E}(G)|$ is the number of sample rows from G output by expression $\mathcal{E}$.*

In summary, we colloquially mention three novel aspects of our accuracy analysis; the details are in the Appendix. Note that prior work [35] applies for `SUM`-like aggregates, uniformly-random samplers and uses self-joins to compute variance. In contrast, `ASALQA` handles different aggregate types including the case when the answer has multiple aggregates. Next, we compute all relevant error measures in one effective pass over data. Finally, to analyze a more general class of samplers, we use two ideas. Any sampler that is strictly more likely to pass a row relative to some uniformly random sampler is analyzable. Further, any sampler that has equivalent error when convolved with all database operators is also analyzable. Our distinct and universe samplers fall into each category respectively. We believe that other samplers exist in each category. These intuitions lead to our definition of sampler dominance.

5. EVALUATION

We have implemented Quickr's samplers and query optimization and deployed it in our production clusters. Here, we present results to answer the following questions:

- How much do queries speed-up? Our evaluation is relative to a version of the QO that is identical in all respects to Quickr except for samplers. We also compare with a state-of-the-art input sampling technique–BlinkDB [9].
- How often are the output plans correct? Every query that receives a sampled plan should meet its error guarantee (with high probability, no groups are missed in the answer and all aggregates are within a small ratio of true value). Further, unapproximable queries should receive a plan with no samplers.
- Where do the gains come from? And, how may these results translate to other queries?



(a) Comparing runtime and resources used **(b) Quantifying error** **(c) Correlating perf. gains to aspects of queries**

Figure 8: Comparing Quickr with Baseline– the production QO in our cluster that is identical to Quickr except for samplers

5.1 Methodology

Comparables: Both of the systems that we compare Quickr against are state-of-the-art. We refer to the production QO without samplers as **Baseline**. Thousands of man-years have gone into developing Baseline, and it is hardened from several years of production use. Baseline supports almost the entirety of T-SQL, has extensive support for UDOs, generates parallel plans and has carefully tuned parallel implementations of many operators. The authors of BlinkDB shared their MILP algorithm to choose which samples to construct. BlinkDB's logic to match a query to available input samples requires queries to recur in order to build the error-latency-profile. We give BlinkDB the benefit of perfect matching by running each query on all of the input samples and use the "best" sample.

Query sets: To share results publicly, we use the queries and datasets from the TPC-DS benchmark [4]. The results here are from a 500GB dataset (scale factor: 500). Results for the 2TB dataset were similar. Table 3 shows some query characteristics. Comparing with Figure 2b we see that TPC-DS queries are simpler than the queries in our cluster– they have fewer passes over data, fewer joins, and smaller QCS. Table 9 reports similar measures for queries from a variety of benchmarks (these queries were run on a Hive cluster) and shows that queries in other benchmarks are even simpler. We chose TPC-DS because it is closest to the workload on our cluster.

Performance Metrics: To compare performance, we measure query runtime as well as the usages of various resources. **Machine-hours** is the sum of the runtime of all tasks. This translates to cluster occupancy and is a measure of throughput. **Intermediate Data** is the sum of the output of all tasks less the job output, i.e., corresponds to the intermediate data written. **Shuffled Data** is the sum of data that moves along the network across racks. Together, the last two measure the excess IO footprint of the query.

Error Metrics: By **Missed Groups**, we refer to the fraction of groups in the answer that are missed. **Aggregation Error** denotes the average error between the estimated and true value of all aggregations in a query. We compute these metrics by analyzing the query output.

Cluster: We evaluate Quickr on the same cluster that ran the production queries that were analyzed in Figure 2b. The servers in this cluster are *datacenter-standard*. That is, each has about ten cores, roughly 100GB of memory, a couple of high RPM disks, some SSDs and a couple of 10Gbps NICs.

5.2 Performance gains

Figure 8a plots a CDF of the various performance metrics. The x axis is a ratio between the values of Baseline and Quickr. An x-value of 2 represents 2× improvement whereas those below 1 represent a regression. We see from the line with star points ("machine-hours") that Quickr lowers resource usage of the median query by over 2×. The improvements in runtime, the line with x points, can be more or less than that of the resource usage due to two factors.

Metric	**Percentile value**					
	10th	25th	50th	75th	90th	95th
# of passes	1.12	1.18	1.3	1.53	1.92	2.61
Total/First pass time	1.26	1.44	1.67	2	2.63	3.42
# Aggregation Ops.	1	1	3	4	8	16
# Joins	2	3	4	7	9	10
depth of operators	17	18	20	23	26	27
# operators	20	23	32	44	52	86
size of QCS + QVS	2	4	5	7	12	17
size of QCS	0	1	3	5	9	11
# user-defined func.	1	2	4	9	14	24

Table 3: Characteristics of the TPC-DS queries used in evaluation.

(1) Queries that process a small amount of data such as TPC-DS q22 are critical-path limited. That is, since our cluster offers much more degree-of-parallelism than needed by some queries, their runtime equals the sum of duration of tasks on the critical path. (2) Other queries gain in runtime by more than expected. When the work to be done reduces, as it does with sampled plans, schedulers can marshall their tasks better. Overall, we find that query runtime improves by a median 1.6×. Roughly 20% of the queries speed up by over 3×. A handful speed up by over 6×. Please note however that roughly 20% of the queries have slightly longer runtimes. This is primarily because (a) query runtime is influenced by task failures and outliers [11, 41] and (b) our cluster scheduler uses fair-sharing and hence the resources offered to queries vary substantially from one execution to another [8, 26]. Regression in machine-hours happens less often and to a smaller degree; this is mostly because our samplers are in C#, but the rest of the query executes in C++ which leads to needless overhead in converting rows from one language format to another. In general, we find runtime to be a noisy metric on our cluster and believe machine-hours which is akin to job makespan to be a more stable metric to compare plan efficiency.

We now focus on some reasons behind these gains. The line with circular points ("interm. data") shows that the total amount of intermediate data written by sampled plans is much smaller. Almost 40% of the queries have reductions larger than 4×. For about 50% of the queries, the intermediate data written is about the same and some plans write even more data. The line with triangle points ("shuffled data") shows that the shuffled volume does not decrease as much as intermediate data. Two facts explain these findings. (1) Quickr triggers parallel plan improvements that build upon the reduced cardinality due to samplers (§A). That is, when the data in flight becomes small, Quickr decreases degree-of-parallelism (DOP) so that hash joins are replaced with cheaper broadcast joins etc. The benefit is faster completion time. The cost, however, is that more data has to be shuffled and written to adjust the DOP (§A). (2) Quickr outputs plans *without samplers* for roughly 25% of the queries. These queries still receive a limited benefit (an average of 25%) because Quickr triggers parallel plan improvements when cardinality reduces due to other aspects such as filters. Such queries will shuffle more data.

To summarize, Quickr offers substantial performance gains to a large fraction of the complex queries in the TPC-DS benchmark.

Metric	Percentile value					
	10th	25th	50th	75th	90th	95th
Baseline QO time	0.38	0.49	0.51	0.54	0.56	0.57
Quickr QO time	0.48	0.5	0.52	0.55	0.57	0.58

Table 4: Query Optimization (QO) times (sec.)

Metric	Value					
	0	1	2	3	4	9
Samplers per query	25%	51%	9%	11%	2%	2%
Sampler-Source dist.	60%	12%	10%	17%	0%	0%

Table 5: Number of samplers per query and their locations

Since production queries have many more passes over data, we observe larger gains overall but defer those results for future work.

5.3 Quantifying error

Figure 8b plots a CDF of the error metrics over all the examined TPC-DS queries. The line with star points ("Missed Groups") shows that up to 20% of queries have missing groups. Upon careful examination, we find that every one of these cases is due to applying `LIMIT 100` on the answer after sorting over the aggregation column. Errors in aggregates change the rank of groups and hence the approximate answer picks a different top 100. We acknowledge that Quickr should more carefully consider this case. To the best of our knowledge, none of the prior AQP schemes handle this scenario either. We make one change to the queries: output the answer before `LIMIT 100` and call that the *full* answer. The line with triangle points ("Missed Groups: Full") shows that Quickr misses no groups in full answers (line is a point on the top left). This is indeed the desired behavior.

In Figure 8b, the lines with circle and x points ("Agg. Error" and "Agg. Error: Full") depict the error for aggregates. We see that 80% of the queries have error within ±10% and over 90% are within ±20%. This is quite good. Careful examination of the outliers reveals two prevalent causes. (a) Support is skewed across groups. Since Quickr assumes an even distribution except for heavy hitters, some groups receive fewer rows than desired and have high error. (b) `SUM`s that are computed over values with high skew have high error. As discussed in §4.1.1, the fix is to stratify on such columns. However, a complication is that value skew changes when passing through predicates. For example, `WHERE X` $> 10^6 \vee$ `X` $< 10^{-3}$ increases skew and `WHERE X` $\in$ `[10, 11]` decreases it. We are working towards *deriving* the value skew statistic. If both of these causes are fixed, and we believe they can be, over 95% of the TPC-DS queries will have aggregates within ±10% of true answer.

5.4 Characterizing what Quickr does

Figure 8c correlates the performance gains with query characteristics. It shows the average metric value for a range in the machine-hours gains. The line with circle points ("Sampler-Source dist.") shows that the gains increase when the samplers are closer to the sources. We next compare some query aspects between the baseline plans and Quickr. The line with x points ("Total/First pass time") and the one with star points ("# of passes") show that the gains due to Quickr are larger for deeper queries. Finally, the line with triangle points ("Interm. Data/10") shows that queries that gain most by Quickr have substantial reductions in intermediate data (up to 19×, since the graph shows values divided by 10).

Table 4 shows the query optimization time for Quickr and Baseline. We expect longer QO times for Quickr since it considers samplers natively and hence explores more alternatives. We ran each query three times and pick the median QO time. The table shows that the increase in QO latency is below 0.1 seconds.

Table 5 shows that 51% of the queries have exactly one sampler. Many have multiple samplers. Further, 25% of the queries are unapproximable. The table also shows where the samplers are in the query plan. "Sampler-Source distance" is the number of IO passes between the extraction stage and the sampler. We see that 60% of the samplers are on the first pass on data. Recall that sampling on the first pass is likely to improve performance the most. We also see that Quickr places samplers in the middle of the plan in many queries; moving such samplers past the next database operator does not yield a better performance vs error tradeoff.

Storage Budget	**Coverage**	**Median Perf. gain: All**	**Median Perf. gain: Covered**	**Median Error**
Default parameters (specifically, K=M=10^5).				
0.5×	0/64	0%	–	–
1×	0/64	0%	–	–
4×	9/64	0%	27%	6%
10×	14/64	0%	24%	5%
Tuned for small group size (K=M=10^1).				
0.5×	8/64	0%	35%	6%
1×	7/64	0%	35%	6%
4×	11/64	0%	32%	6%
10×	12/64	0%	24%	6%

Table 6: BlinkDB's performance on TPC-DS.

Table 7 shows how often various samplers are used. Overall, uniform sampler is used roughly twice as frequently as the distinct and universe samplers. The distinct sampler is often replaced by the uniform because when there is enough support for groups, the latter is accurate enough but has better performance. Analogously, the universe sampler is often optimized away because it is dominated by both the uniform and the distinct samplers (see Proposition 6 in §B.3). Quickr uses the universe sampler only for queries that join two large relations. This happens more often in our production queries. We note that even complex queries with multiple joins receive plans with only a few samplers. A key reason is that Quickr converts stratification requirements on columns from the smaller relations to stratification over join keys in the larger relations (see discussion on sfm in §4.2.4). Without our sampling+QO logic (§4.2), *evaluating* the accuracy and performance of the many possible sampled plans requires complex reasoning and is time-consuming even for an experienced data scientist.

5.5 Quickr vs. Apriori samples

The fundamental problem with apriori samples is the poor coverage for any feasible storage budget. That is, when queries are rich, tables have many columns and a dataset has many tables, any choice of differently stratified samples of the inputs has poor coverage for feasible storage budgets. To illustrate this aspect, we evaluate BlinkDB [9], the best exemplar system of this approach, on the TPC-DS benchmark. BlinkDB's MILP to decide which samples to obtain works only for one table and extending to multiple tables is non-trivial due to reasons described below. We use the MILP to generate samples for the `store_sales` table because (a) it is the largest table in the TPC-DS benchmark and (b) it has the highest potential to improve query performance; out of the 64 queries that we consider, 40 use the `store_sales` table. Next, we run each query on *all* of the samples chosen by the MILP and pick the sample with the best possible performance that meets the error constraint (no groups missed and less than ±10% error for aggregates). We vetted our methodology and the results with the authors of BlinkDB.

Table 6 shows our findings when using the same parameter values as in the BlinkDB paper. We ran the queries in Hive [40] atop Tez [2]. The samples are not explicitly stored in memory but the file-system cache does help since no other queries ran on the cluster during our experiment. We see that very few queries benefit. For most queries, none of the constructed input samples yield an answer with zero missed rows and within ±10% error on aggregates.

For sample-set sizes equal to or smaller than the input, the best coverage was 13%. A parameter sweep on BlinkDB's internal parameters reveals that the best coverage overall was 22%, obtained at 10× the size of the input. Of the 14 queries that benefited, the median speed-up was 24%. It is more feasible to store the entire input in memory than such large sample-sets.

Digging a bit further, we find that BlinkDB's MILP generates over 20 differently stratified samples on `store_sales`. Most are on one column but several are on column pairs. Many TPC-DS queries have a large QCS (see Table 3); large QCSets have a large sample size and hence samples on column-pairs were picked by the MILP only at high storage budgets.

We take care to point out that apriori samples can be useful. Predictable aggregation queries that touch only one large dataset will greatly benefit. Especially if the data distribution is sparse, i.e. the number of distinct values per group is much smaller than the row count. Then, each stratified sample will be small and many different stratified samples can be feasibly stored per popular dataset. However, this does not happen in TPC-DS or in our production clusters. Further complications include: keeping samples consistent when datasets churn continuously and choosing which among the available samples (if any) would help an ad-hoc query that differs from the queries used when constructing samples. Storing apriori samples for queries that join more than one large table is problematic because the same tables can be joined on multiple columns and the samples constructed for a particular join-of-tables do not help queries that join in a different manner (for example, nine queries join `store_sales` and `store_returns` in TPC-DS with four different options for join columns).

6. RELATED WORK

Many big data systems support relational queries [12, 18, 33, 37, 40]. Several offer a (uniform) sampler operator. But none automatically decide which queries can be sampled, place appropriate samplers at appropriate locations or offer guarantees on answer quality.

A rich vein of literature samples input datasets. See [24] for an excellent overview. Some update the samples as datasets evolve [6]. Most assume knowledge of the queries and the datasets. Congressional sampling [7] keeps both uniform and stratified samples on the group-by columns used by queries. STRAT [20] computes the optimal sample-set to store given a budget. Chaudhuri et.al. [19] maintains an *outlier index* to better support skew. Babcock et. al. [14] stores the more uncommon values explicitly. SciBORQ [39] stores multiple layers of *impressions* where each layer can have a different focus and level of detail. Similar to STRAT, BlinkDB [9] also optimally chooses stratified samples. Uniquely, it stores samples in memory and computes an error-latency-profile for repetitive queries to meet error or performance targets. For complex ad-hoc queries with a heavy-tailed distribution on inputs, apriori sampling has substantial shortcomings (see §5.5).

Online aggregation (OLA) [29, 30, 43] progressively processes more of the input and keeps updating the answer. The use-case is exciting since the user can run the query until the answer is satisfactory. Quickr looks at all tuples once (early in the plan) and retains a small subset for the remainder of the query. Hence, Quickr can offer good error bounds without any assumptions on physical layout (e.g., input being randomly ordered) and with less computational overhead (e.g., does not need bootstrap). Further, OLA requires specialized join operators and in-memory data-structures to efficiently update the answers of a complex query [23, 38].

When sampling, Quickr does not use existing indices on tables as well as some others do [24, 36]. While this choice makes Quickr more widely applicable, for example on unstructured data or when no index exists, leveraging such metadata can make Quickr more effective. For example, by sampling blocks or sub-trees of a B-tree instead of examining every tuple. This is key future work.

Our accuracy analysis technique (§4.3) improves in three ways over the closest related technique [35]. First, the previous technique [35] only applies for generalized uniform samplers (GUS). Neither our universe nor our distinct sampler belong to the class of GUS. We offer a new notion of sampler dominance that is a significant generalization. Second, we also analyze the likelihood of missing groups and consider aggregates beyond SUM (see §B, §C). Finally, while [35] requires self-joins to compute the variance, Quickr computes variance and other error metrics in effectively one pass (§4.3).

Recently, there has been some work in using bootstrap to determine which apriori sample is suited for a query [10]. Since bootstrap can need thousands of trials, each of which resample from a sample [44], recent work shows how to symbolically execute bootstrap (by using probability annotations and new relational operators that convolve random variables) [44]. However, neither offset the shortcomings of apriori samples. Since Quickr samples in-line, our accuracy analysis is simpler, faster and has good guarantees.

Our uniform sampler (§4.1.1) is the standard Poisson sampler [24] used by several prior works. The novelty in our distinct sampler (§4.1.2) derives largely from its execution in a streaming and partitioned manner: specifically in the way we lower bias and reduce memory footprint. To the best of our knowledge, the universe sampler (§4.1.3) is novel. Prior efforts on sampling both inputs of a join are much more complex [6, 21, 22, 24]. They use detailed statistics or indices on one or more of the input relations of the join (e.g., Strategy Stream Sample [21]). Since the join inputs in big data queries are intermediate data that is generated on the fly, obtaining such statistics has substantial overhead. It requires an additional pass over data and affects parallelism since the sampler on one input has to wait for the stat collector on the other. Worse, prior work requires samplers to *immediately precede* the join. That is, they cannot be pushed down past other database operations since maintaining the required statistics becomes even more complex. Our universe sampler requires no data exchange between the join inputs at execution time (the hash function and the portion of hash space to be picked are parameters from the query plan). Finally, we are aware of no prior work that reasons about samplers natively in the context of a query optimizer.

7. FINAL THOUGHTS

We offer a new way to approximate complex ad-hoc queries. Our insight is to leverage the many passes over data that such queries perform to sample lazily on the first pass. The ability to sample multiple join inputs precisely and reasoning about samplers natively in a query optimizer allowed us to offer a turn-key solution where the user only needs to submit a query and the system would automatically determine if and how best that query can be approximated. Initial results are very promising. In regimes where previously known techniques deliver no gains, Quickr offers substantial speedup. The speed-up often translates to lower cost (e.g., when using public clouds), faster results or the ability to support many more queries without increasing cluster size by up to 2×. We also extend the state-of-art in analyzing the error of sampled expressions.

8. ACKNOWLEDGMENTS

We thank Chris Douglas, Alekh Jindal, Konstantinos Karanasos, Suman Nath, Raghu Ramakrishnan, Jingren Zhou and the Cosmos team at Microsoft for their feedback, advice and support. This draft also benefited from feedback from the SIGMOD reviewers.

9. REFERENCES

[1] AMPLab BigData Benchmark. http://bit.ly/1uyuBE8.

[2] Apache Tez. http://tez.apache.org/.

[3] Intel Big-Data-Benchmark. http://bit.ly/1HlFRH0.

[4] TPC-DS Benchmark. http://bit.ly/1J6uDap.

[5] TPC-H Benchmark. http://bit.ly/1KRK5gl.

[6] S. Acharya et al. The aqua approximate query answering system. In *SIGMOD*, 1999.

[7] S. Acharya et al. Congressional samples for approximate answering of group-by queries. In *SIGMOD*, 2000.

[8] S. Agarwal et al. Re-optimizing data parallel computing. In *NSDI*, 2012.

[9] S. Agarwal et al. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *EuroSys*, 2013.

[10] S. Agarwal et al. Knowing when you're wrong: Building fast and reliable approximate query processing systems. In *SIGMOD*, 2014.

[11] G. Ananthanarayanan et al. Reining in the Outliers in MapReduce Clusters Using Mantri. In *OSDI*, 2010.

[12] M. Armbrust et al. Spark sql: Relational data processing in spark. In *SIGMOD*, 2015.

[13] B. Babcock et al. Models and issues in data stream systems. *PODS*, 2002.

[14] B. Babcock et al. Dynamic sample selection for approximate query processing. In *SIGMOD*, 2003.

[15] Z. Bar-Yossef et al. Counting distinct elements in a data stream. In *RANDOM*, 2002.

[16] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinctvalue estimation under multiset operations. In *SIGMOD*, 2007.

[17] J. Brutlag. Speed matters for Google web search. http://bit.ly/1b4RKoZ, 2009.

[18] R. Chaiken et al. SCOPE: Easy and Efficient Parallel Processing of Massive Datasets. In *VLDB*, 2008.

[19] S. Chaudhuri et al. Overcoming limitations of sampling for aggregation queries. In *ICDE*, 2001.

[20] S. Chaudhuri et al. A robust optimization-based approach for approximate answering of aggregate queries. *SIGMOD*, 2001.

[21] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, 1999.

[22] S. Chaudhuri, R. Motwani, and V. R. Narasayya. Random sampling for histogram construction: How much is enough? In *SIGMOD 1998*, 1998.

[23] T. Condie et al. Mapreduce online. In *NSDI*, 2010.

[24] G. Cormode et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 2012.

[25] J. R. Dabrowski and E. V. Munson. Is 100 Milliseconds Too Fast? In *CHI*, 2001.

[26] A. Ferguson et al. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *EuroSys*, 2012.

[27] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 1995.

[28] A. Halevy. Answering queries using views: A survey. 2001.

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

[30] C. M. Jermaine et al. Scalable approximate query processing with the DBO engine. In *SIGMOD*, 2007.

[31] T. Johnson et al. Sampling algorithms in a stream operator. In *SIGMOD*, 2005.

[32] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *VLDB*, 2002.

[33] S. Melnik et al. Dremel: Interactive analysis of web-scale datasets. In *VLDB*, 2010.

[34] R. Motwani et al. Query processing, resource management, and approximation in a data stream management system. *CIDR*, 2003.

[35] S. Nirkhiwale, A. Dobra, and C. Jermaine. A sampling algebra for aggregate estimation. In *PVLDB*, 2013.

[36] F. Olken. *Random Sampling from Databases*. PhD thesis, UCBerkeley, 1993.

[37] C. Olston et al. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*, 2008.

[38] N. Pansare et al. Online aggregation for large mapreduce jobs. In *VLDB*, 2011.

[39] L. Sidirourgos et al. Sciborq: Scientific data management with bounds on runtime and quality. In *CIDR*, 2011.

[40] A. Thusoo et al. Hive- a warehousing solution over a map-reduce framework. In *VLDB*, 2009.

[41] Y. Kwon, M. Balazinska, B. Howe, J. Rolia. Skew-Resistant Parallel Processing of Feature-Extracting Scientific User-Defined Functions. In *SOCC*, 2010.

[42] Y. Yan et al. Error-bounded sampling for analytics on big sparse data. In *VLDB*, 2014.

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

[44] K. Zeng et al. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *SIGMOD*, 2014.

APPENDIX

A. FURTHER DETAILS OF ASALQA

In §4.2.6, we see how Quickr translates each logical sampler to a physical implementation. Next, the QO costs sampled subexpressions. To enable this, we make the following changes: (1) Assign the correct cardinality for the samplers based on their sample probability p except for the distinct sampler which will leak more rows due to the requirement to pass at least δ rows for each distinct value of strat cols. (2) Assign the correct processing costs to the samplers per input and output row. The uniform sampler is the most efficient since it only tosses a coin per row. The universe sampler comes next since it uses a cryptographic hash. The distinct sampler has the highest processing cost because it invokes the heavy-hitter sketch and uses more memory for both the sketch and the reservoirs.

Sampler→{Sampler, Exchange}: Reducing the degree-of-parallelism (DOP) is necessary to get more gains from samplers. To see why, say a sampler reduces the number of rows in a relation by 10×, the work does become $\frac{1}{10}$'th now but start-up costs and other overheads remain the same. Reducing the DOP amortizes this overhead and triggers further improvements later in the plan. For example, hash joins can be replaced with cross joins, re-partitions can be avoided and parallel plans can be replaced with a serial plan when data in flight is small [8]. Quickr introduces exchange operators to reduce the DOP. However, since an exchange shuffles data it increases the cost (in terms of latency and resources to shuffle) and hence ASALQA places an exchange only when the cost is smaller than the gains in subsequent portions of the execution plan.

Global requirements and Caveats: To ensure that both sides of a join are implemented with the same universe sampler parameters (if

Metric	Sampler Type		
	Uniform	Distinct	Universe
Distribution across samplers	54%	26%	20%
Queries that use at least 1 sampler of a certain type	49%	24%	9%

Table 7: Frequency of use of various samplers.

True value		Estimate rewritten by Quickr	
`SUM(X)`	`COUNT(*)`	`SUM(`$w\cdot$` X)`	`SUM(`w`)`
`AVG(X)`		`SUM(`$w\cdot$`X)/SUM(`w`)`	
`SUM(IF(F(X)? Y: Z))`		`SUM(IF(F(X)? `$w\cdot$` Y: `$w\cdot$` Z))`‡	
`COUNT(DISTINCT X)`		`COUNT(DISTINCT X)·(univ(X)? `w`:1)`	

‡ : COUNTIF and COUNT(X) are rewritten analogous to SUMIF.

Table 8: How Quickr rewrites aggregation operations.

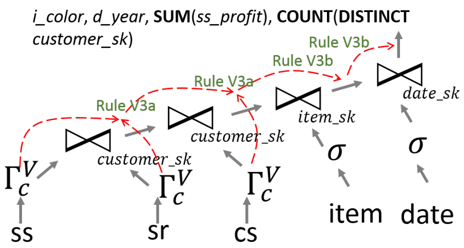


Figure 9: For the example in Figure 1, the query plan computed by ASALQA. The dashed arrows show how our error analysis unrolls ASALQA and the dominance rules (§B.3) used at each step. The result is an expression with a single sampler just below the aggregation.

Metric	Percentile values					
	50th			90th		
Total/First pass time	1.2	1.4	1.3	2.8	4.3	2
# of Passes over Data	3.1	1.1	1.0	4.0	1.2	1.3
# Aggregation Ops.	3	1	2	8	3	5
# Joins	4	2	2	9	5	7
depth of operators	20	18	16	26	24	27
size of QCS+QVS	5	5	5	12	9	4
size of QCS	3	3	4	9	8	3

Table 9: Analyzing query attributes from different workloads: TPC-DS [4], TPC-H [5] and Other (BigBench [3] ∪ BigData [1] ∪ . . .). Queries are written in Hive and executed with Tez as the AM in Yarn.

at all), on the bottom-up pass of the query optimization, ASALQA rejects plans that do not satisfy such *global* requirements. It then falls back to the next best choice. Further, Quickr does not allow nested samplers. Because the performance gains from sampling a relation that has been sampled already is not worth the added potential for error. This too is implemented during the bottom-up pass. Finally, we admit that a few aspects of Quickr are as yet unimplemented. Quickr does not push samplers past spools and full outer joins. Heavy hitter information is not used as well as it should be. And, though we obtain column value variance at the inputs, we do not yet derive the value variance past other operations. In our evaluation, we point out the effect of these caveats. The first two lower Quickr's coverage and performance gains– fewer queries benefit and by less than they should– and the last adds to error when computing SUM over skewed columns. Our results are already quite good, however.

B. ANALYZING SAMPLED PLANS

We formally analyze our samplers and push-down rules. We focus here on queries that perform SUM-like aggregations over groups; other aggregates are covered in §C. Let group G denote all the tuples that have the same value in group-by columns of the output. The answer contains for each group G the aggregate $w(G) = \sum_{t\in G} w(t)$, where $w(t)$ is the value associated with tuple t. Given a query E, Quickr's ASALQA algorithm outputs a query plan $\mathcal{E}$ with samplers at various locations. Our goal is to compare the answer of E (unsampled query) with that of $\mathcal{E}$ on two aspects: the *expected squared error* of group values (i.e., variance of $w(G)$) and the *group coverage probability* (i.e., likelihood of not missing a group).

For all three samplers used by Quickr, we first offer closed-form expressions upon immediate use. That is, when the samplers are placed near at the root of the query plan just before group-by and aggregation (see §B.1). To compute similar expressions when samplers are at arbitrary locations in the query tree, intuitively, we address the following issue: when pushing a sampler past another database operator, how do the error expressions change? (See §B.2.) We introduce the concept of *sampling dominance* between query expressions which ensures that the error is no worse. Using this, we establish in §B.3 a collection of sampler transformation rules. which guide our exploration of alternatives as well as our accuracy analysis.

B.1 Estimating Sampler Accuracy

Quickr uses the *Horvitz-Thompson (HT) estimator* [24] to relate the answers on sampled query plans to their true values and to estimate the expected squared error. For each group G in the answer of an unsampled query E, the sampled plan $\mathcal{E}$ outputs a subset of the rows in G, $\mathcal{E}(G) \subseteq G$. We estimate $w(G)$ as:

$$\hat{w}_{\mathcal{E}}(G) = \sum_{t\in\mathcal{E}(G)} \frac{w(t)}{\mathbf{Pr}\left[t \in \mathcal{E}(G)\right]}. \tag{1}$$

It is easy to see that the above (HT) estimator is unbiased, i.e., $\mathbf{E}\left[\hat{w}_{\mathcal{E}}(G)\right] = w(G)$. Hence, its variance is *expected squared error*:

$$\mathbf{Var}\left[\hat{w}_{\mathcal{E}}(G)\right] = \sum_{i,j\in G}\left(\frac{\mathbf{Pr}\left[i,j \in \mathcal{E}(G)\right]}{\mathbf{Pr}\left[i \in \mathcal{E}(G)\right]\mathbf{Pr}\left[j \in \mathcal{E}(G)\right]} - 1\right)\cdot w(i)w(j).$$

From the sample $\mathcal{E}(G)$, $\mathbf{Var}\left[\hat{w}_{\mathcal{E}}(G)\right]$ can be estimated as:

$$\hat{\mathbf{Var}}\left[\hat{w}_{\mathcal{E}}(G)\right] = \tag{2}$$

$$\sum_{i,j\in\mathcal{E}(G)}\left(\frac{\mathbf{Pr}\left[i,j \in \mathcal{E}(G)\right]}{\mathbf{Pr}\left[i \in \mathcal{E}(G)\right]\mathbf{Pr}\left[j \in \mathcal{E}(G)\right]} - 1\right)\cdot \frac{w(i)w(j)}{\mathbf{Pr}\left[i,j \in \mathcal{E}(G)\right]}.$$

Recall our three samplers: *uniform sampler* Γ^{U}_p (uniform sampling probability p), *distinct sampler* $\Gamma^{\mathsf{D}}_{p,\mathcal{C},\delta}$ (each value of column set $\mathcal{C}$ has support at least δ in the sample), and *universe sampler* $\Gamma^{\mathsf{V}}_{p,\mathcal{C}}$ (sampling values of column set $\mathcal{C}$ with probability p). It could help to think of the universe sampler as a predicate that passes only the rows whose values of $\mathcal{C}$ belong to the chosen subspace.

We apply the HT estimator to compute variance for all the samplers. To do so, we compute the terms $\mathbf{Pr}\left[i \in \mathcal{E}(G)\right]$ and $\mathbf{Pr}\left[i,j \in \mathcal{E}(G)\right]$ for each sampler as follows:

Proposition 3 (To Compute HT Estimator and the Variance)**.**

- *For Γ^{U}_p, for any tuples $i, j \in G$, we have $\mathbf{Pr}\left[i \in \mathcal{E}(G)\right] = p$, and, if $i \neq j$, $\mathbf{Pr}\left[i,j \in \mathcal{E}(G)\right] = p^2$.*
- *For $\Gamma^{\mathsf{D}}_{p,\mathcal{C},\delta}$, let $g(i)$ be the set of tuples with the same values on $\mathcal{C}$ as tuple i in the input relation. We have*

  $$\mathbf{Pr}\left[i \in \mathcal{E}(G)\right] = \begin{cases} 1 & |g(i)| \leq \delta \\ \max(\delta/|g(i)|, p) & |g(i)| > \delta \end{cases};$$

  if $i \neq j$, $\mathbf{Pr}\left[i,j \in \mathcal{E}(G)\right] = \mathbf{Pr}\left[i \in \mathcal{E}(G)\right]\mathbf{Pr}\left[j \in \mathcal{E}(G)\right]$.
- *For $\Gamma^{\mathsf{V}}_{p,\mathcal{C}}$, let $g(i)$ be the set of tuples with the same values on $\mathcal{C}$ as i in the input relation. We have $\mathbf{Pr}\left[i \in \mathcal{E}(G)\right] = p$, and*

  $$\mathbf{Pr}\left[i,j \in \mathcal{E}(G)\right] = \begin{cases} p & g(i) = g(j) \\ p^2 & g(i) \neq g(j) \end{cases}, \text{ if } i \neq j.$$

A few imprecisions are worth mentioning. The universe sampler does choose its subspace at random. So while the formula above is technically accurate, tuples that belong to the subspace will be in the sample and those outside the subspace would not. Our implementation of the distinct sampler, because of its requirements to finish in one pass and with a small memory footprint, introduces some correlation between tuples and a slight bias as noted already.

Complexity of Computing Estimate and Error.
Proposition 2 posits that the computation requires only one effective pass over the sample. The proof follows from Proposition 3 and Equations(1)-(2). Since each tuple i in the sample $\mathcal{E}(G)$ also contains the probability $\mathbf{Pr}\left[i \in \mathcal{E}(G)\right]$ in its weight column, $\hat{w}_{\mathcal{E}}(G)$ can be computed in one scan using Equation(1). A naive way to compute $\hat{\mathbf{Var}}\left[\hat{w}_{\mathcal{E}}(G)\right]$ using (2) requires a self-join and can take quadratic time since it checks all pairs of tuples in the sample. We do better by observing that only the pairs having $\mathbf{Pr}\left[i, j \in \mathcal{E}(G)\right] \neq \mathbf{Pr}\left[i \in \mathcal{E}(G)\right] \mathbf{Pr}\left[j \in \mathcal{E}(G)\right]$ need to be considered. For the uniform and distinct samplers, the summation in (2) goes to zero for $i \neq j$ and so their variance can be computed in one pass. For the universe sampler, there are two types of pairs: i) (i, j) with $g(i) = g(j)$, and ii) (i, j) with $g(i) \neq g(j)$. Per Proposition 3, the summation term is zero for pairs of the latter type. For the former type, we maintain per-group values in parallel and use a shuffle to put them back into (2). Since the number of groups can be no larger than the number of tuples, the computation is linear. Further the shuffle often has much less work to do (per group) than the first pass (per sampled tuple) leading to our one-effective-pass claim.

Group Coverage Probability.
A group G will miss from the answer if all the tuples in G are missed in the sample $\mathcal{E}(G)$. We show how Quickr makes this unlikely.

Proposition 4 (Group Coverage Probability)**.** *When samplers immediately precede the aggregate, the probability that a group G appears in the answer is:*

- *For* Γ_p^{U}, $\mathbf{Pr}\left[G\right] = 1 - (1-p)^{|G|}$.
- *For* $\Gamma_{p,\mathcal{C},\delta}^{\mathsf{D}}$,
$$\mathbf{Pr}\left[G\right] \begin{cases} = 1, & \textit{if } \mathcal{C} \textit{ contains the group-by dimensions} \\ \geq 1 - (1-p)^{|G|}, & \textit{otherwise} \end{cases}.$$
- *For* $\Gamma_{p,\mathcal{C}}^{\mathsf{V}}$, $\mathbf{Pr}\left[G\right] = 1 - (1-p)^{|G(\mathcal{C})|}$*, where* $G(\mathcal{C})$ *is the set of distinct values of tuples in G on dimensions* $\mathcal{C}$*.*

Using Proposition 4, we see that both uniform and distinct samplers rarely miss groups. Recall that Quickr checks before introducing samplers that there is enough support, i.e., $p * |G| \geq k$ (§4.2.6). For example, when $k = 30$ and $p = 0.1$, the likelihood of missing G is below 10^{-14}. For the universe sampler, note that $|G(\mathcal{C})| \in [1, |G|]$. However, recall that Quickr uses the universe sampler only when the stratification requirements can be met. That is, the overlap between universe columns and those defining the group is small. Hence, $|G(\mathcal{C})| \sim |G|$ and groups are missed rarely.

Sampling from Join.
The advantage of universe sampler $\Gamma_{p,\mathcal{C}}^{\mathsf{V}}$ lies in sampling from join. To draw a p fraction of tuples from the join of two relations, the universe sampler only needs a p fraction from input relations. But, both the uniform and distinct samplers need to draw a $\sqrt{p}$ fraction of tuples. Not only is the $\sqrt{p}$ sample more expensive to compute but both the variance and the group coverage probability become worse (when p is replaced with $\sqrt{p}$ in Propositions 3 and 4).

B.2 Sampling Dominance

We now formalize our notion of sampling dominance between query expressions. Suppose $\mathcal{E}$ is an expression on database relations possibly with samplers. The *core* $\Lambda(\mathcal{E})$ denotes the expression generated by removing all samplers from $\mathcal{E}$. We say an expression $\mathcal{E}_1$ is dominated by another expression $\mathcal{E}_2$ if and only if the expressions have the same core and $\mathcal{E}_2$ has no higher variance and no higher probability of missing groups than $\mathcal{E}_1$. More formally, we have:

Definition 1 (Sampling Dominance)**.** *Given two expressions* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *with* $\Lambda(\mathcal{E}_1) = \Lambda(\mathcal{E}_2)$ *and having* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *as the respective output relations, we say* $\mathcal{E}_2$ dominates $\mathcal{E}_1$*, or* $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$*, iff*

$$(\textit{v-dominance } \mathcal{E}_1 \overset{\mathsf{v}}{\Rightarrow} \mathcal{E}_2) \quad \forall i, j: \tag{3}$$
$$\frac{\mathbf{Pr}\left[i \in \mathcal{R}_1, j \in \mathcal{R}_1\right]}{\mathbf{Pr}\left[i \in \mathcal{R}_1\right]\mathbf{Pr}\left[j \in \mathcal{R}_1\right]} \geq \frac{\mathbf{Pr}\left[i \in \mathcal{R}_2, j \in \mathcal{R}_2\right]}{\mathbf{Pr}\left[i \in \mathcal{R}_2\right]\mathbf{Pr}\left[j \in \mathcal{R}_2\right]}, \textit{ and}$$
$$(\textit{c-dominance } \mathcal{E}_1 \overset{\mathsf{c}}{\Rightarrow} \mathcal{E}_2) \quad \forall t: \mathbf{Pr}\left[t \in \mathcal{R}_1\right] \leq \mathbf{Pr}\left[t \in \mathcal{R}_2\right]. \tag{4}$$

Note that *sampler dominance* subsumes the *SOA-equivalence* definition from [35]. Two expressions $\mathcal{E}_1, \mathcal{E}_2$ are SOA equivalent iff $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$ and $\mathcal{E}_2 \overset{*}{\Rightarrow} \mathcal{E}_1$. Intuitively c-dominance says that all tuples are strictly more likely to appear in the output and v-dominance helps relate the variance. By using (3) and (4) in (1) and the above propositions, it is not hard to see that if $\mathcal{E}_2$ dominates $\mathcal{E}_1$, $\hat{w}_{\mathcal{E}_2}(G)$ is better than $\hat{w}_{\mathcal{E}_1}(G)$ in terms of variance and group coverage probability. We formally state this result below.

Proposition 5 (Dominance and Accuracy)**.** *For any group G in the output of a SUM-like aggregate query, consider two execution plans* $\mathcal{E}_1$ *and* $\mathcal{E}_2$ *with independent (uniform, distinct, universe) samplers, with the same core plan* $\Lambda(\mathcal{E}_1) = \Lambda(\mathcal{E}_2)$*. If* $\mathcal{E}_1 \overset{\mathsf{v}}{\Rightarrow} \mathcal{E}_2$*, we have*

$$\mathbf{Var}\left[\hat{w}_{\mathcal{E}_1}(G)\right] \geq \mathbf{Var}\left[\hat{w}_{\mathcal{E}_2}(G)\right].$$

And if $\mathcal{E}_1 \overset{\mathsf{c}}{\Rightarrow} \mathcal{E}_2$ *and the values on the join dimension and the grouping dimension (if any) are sampled independently, we have*

$$\mathbf{Pr}\left[G \textit{ is missed in } \mathcal{E}_1\right] \geq \mathbf{Pr}\left[G \textit{ is missed in } \mathcal{E}_2\right].$$

Hence, $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$, implies that the latter has a strictly better answer.

Dominance Transitivity across Database Operators.
Eventually, we want the dominance relationship to hold at the root of a plan so that we can bound the variance and the group-missing probability in the answer according to Proposition 5. We show that dominance is transitive across database operators, from the root to leaves in the plan. We focus on three operators: π_C (projection on a subset of columns $\mathcal{C}$), σ_C (selection on $\mathcal{C}$), and $\bowtie_C$ (join on $\mathcal{C}$).

This leads us to Proposition 1 which states the conditions under which the dominance relationship is transitive. The proof follows.

Proof. Note that projection holds by definition. We focus on select (ii) and join (iii) below.

Let $\mathcal{R}_i$ be the set of rows output by $\mathcal{E}_i$, and R be the set of rows output by $\Lambda(\mathcal{E}_i)$. For a row $i \in \sigma_C(R)$, we have
$$\mathbf{Pr}\left[i \in \sigma_C(\mathcal{R}_i)\right] = \begin{cases} \mathbf{Pr}\left[i \in \mathcal{R}_i\right] & i \in \sigma_C(R) \\ 0 & i \notin \sigma_C(R) \end{cases}. \tag{5}$$

ii) can be proved by putting (5) into (3) and (4).

To prove $\mathcal{E}_1 \bowtie_C \mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{E}_2 \bowtie_C \mathcal{F}_2$, it suffices to prove $\mathcal{E}_1 \times \mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{E}_2 \times \mathcal{F}_2$ (i.e., cross product) since $\bowtie_C$ is equivalent to composing $\times$ with selection σ_C and we can apply ii) for the latter. Suppose $\mathcal{S}_i$ is the set of rows output by $\mathcal{F}_i$, and S is the set of rows output by $\Lambda(\mathcal{F}_i)$. For $\mathcal{E}_1 \times \mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{E}_2 \times \mathcal{F}_2$ to hold, we need to show that
$$\frac{\mathbf{Pr}\left[(r,s) \in \mathcal{R}_1 \times \mathcal{S}_1, (r',s') \in \mathcal{R}_1 \times \mathcal{S}_1\right]}{\mathbf{Pr}\left[(r,s) \in \mathcal{R}_1 \times \mathcal{S}_1\right]\mathbf{Pr}\left[(r',s') \in \mathcal{R}_1 \times \mathcal{S}_1\right]}$$
$$\geq \frac{\mathbf{Pr}\left[(r,s) \in \mathcal{R}_2 \times \mathcal{S}_2, (r',s') \in \mathcal{R}_2 \times \mathcal{S}_2\right]}{\mathbf{Pr}\left[(r,s) \in \mathcal{R}_2 \times \mathcal{S}_2\right]\mathbf{Pr}\left[(r',s') \in \mathcal{R}_2 \times \mathcal{S}_2\right]}. \tag{6}$$

To this end, we consider the following two cases.

Case a) $\mathcal{E}_i$ and $\mathcal{F}_i$ share a universe sampler $\Gamma^{\mathsf{V}}_{p,\mathcal{D}}$. Let $r_{\mathcal{D}}$ be the value of a row r on dimensions $\mathcal{D}$. In this case, the event "$(r,s) \in \mathcal{R}_i \times \mathcal{S}_i$" is equivalent to that r and s have the same values on dimensions $\mathcal{D}$, i.e., $r_{\mathcal{D}} = s_{\mathcal{D}}$, and the value $r_{\mathcal{D}}$ is picked by the universe sampler. Because, when $r_{\mathcal{D}} \neq s_{\mathcal{D}}$ or $r'_{\mathcal{D}} \neq s'_{\mathcal{D}}$, we have both sides of (6) equal to 0; when $r_{\mathcal{D}} = s_{\mathcal{D}}$ and $r'_{\mathcal{D}} = s'_{\mathcal{D}}$, we have

$$\mathbf{Pr}\left[(r,s) \in \mathcal{R}_i \times \mathcal{S}_i, (r',s') \in \mathcal{R}_i \times \mathcal{S}_i\right] = \mathbf{Pr}\left[r \in \mathcal{R}_i, r' \in \mathcal{R}_i\right]$$
$$\text{and } \mathbf{Pr}\left[(r,s) \in \mathcal{R}_i \times \mathcal{S}_i\right] = \mathbf{Pr}\left[r \in \mathcal{R}_i\right]. \tag{7}$$

From (7) and $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$, $\mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{F}_2$ we have $\mathcal{E}_1 \times \mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{E}_2 \times \mathcal{F}_2$.

Case b) $\mathcal{E}_i$ and $\mathcal{F}_i$ do not share a universe sampler. In this case, we know that samplers on the two sides of the join operator (or ×) are independent, i.e., rows from $\mathcal{R}_i$ and the ones from $\mathcal{S}_i$ are drawn independently. So based on this independence condition, we have

$$\mathbf{Pr}\left[(r,s) \in \mathcal{R}_i \times \mathcal{S}_i, (r',s') \in \mathcal{R}_i \times \mathcal{S}_i\right]$$
$$= \mathbf{Pr}\left[r \in \mathcal{R}_i, r' \in \mathcal{R}_i\right] \cdot \mathbf{Pr}\left[s \in \mathcal{S}_i, s' \in \mathcal{S}_i\right], \text{and} \tag{8}$$
$$\mathbf{Pr}\left[(r,s) \in \mathcal{R}_i \times \mathcal{S}_i\right] = \mathbf{Pr}\left[r \in \mathcal{R}_i\right] \cdot \mathbf{Pr}\left[s \in \mathcal{S}_i\right]. \tag{9}$$

Putting (8) and (9) into (6), we can see that $\mathcal{E}_1 \overset{*}{\Rightarrow} \mathcal{E}_2$ and $\mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{F}_2$ suffice for $\mathcal{E}_1 \times \mathcal{F}_1 \overset{*}{\Rightarrow} \mathcal{E}_2 \times \mathcal{F}_2$. □

B.3 Sampler Switching and Pushing Rules

For the same sampling rate p, we have the following ranking among samplers in the order of accuracy from lower to higher.

Proposition 6 (Switching Rule)**.** *For any relation R, we have* $\Gamma^{\mathsf{V}}_{p,\mathcal{C}}(R) \overset{*}{\Rightarrow} \Gamma^{\mathsf{U}}_{p}(R) \overset{*}{\Rightarrow} \Gamma^{\mathsf{D}}_{p,\mathcal{C},\delta}(R)$.

The proof follows from observing the terms for variance and group coverage of each sampler. Since the distinct sampler has lower performance gain, it is used only when needed for accuracy.

We now list a few useful *transformation* rules based on our dominance definition and transitivity proofs.

Proposition 7 (Pushing past Projection)**.** *For any relation R and a projection $\pi_{\mathcal{C}}$ where $\mathcal{C}$ is a subset of columns of R, we have*

Rule-U1: $\Gamma^{\mathsf{U}}_{p}(\pi_{\mathcal{C}}(R)) \overset{*}{\Rightarrow} \pi_{\mathcal{C}}(\Gamma^{\mathsf{U}}_{p}(R))$;

Rule-D1: $\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(\pi_{\mathcal{C}}(R)) \overset{*}{\Rightarrow} \pi_{\mathcal{C}}(\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(R))$, *if* $\mathcal{D} \subseteq \mathcal{C}$;

Rule-V1: $\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(\pi_{\mathcal{C}}(R)) \overset{*}{\Rightarrow} \pi_{\mathcal{C}}(\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R))$, *if* $\mathcal{D} \subseteq \mathcal{C}$.

The rules show that it is strictly better to push samplers below projects. Indeed, the sampler column set $\mathcal{D}$ is always a subset of the columns returned by the project $\mathcal{C}$.

Pushing samplers past selections is complicated by one aspect. If the columns used in the select are not explicitly stratified, the group sizes vary before and after pushing. Hence, we introduce the *weak dominance relationship*, denoted as $\overset{\sim}{\Rightarrow}$. With weak dominance, v- and c- dominance only hold for large groups in a probabilistic way.

Proposition 8 (Pushing past Selection)**.** *For any relation R and a selection $\sigma_{\mathcal{C}}$ with selection formula on a subset $\mathcal{C}$ of columns of R,*

Rule-U2: $\Gamma^{\mathsf{U}}_{p}(\sigma_{\mathcal{C}}(R)) \overset{*}{\Rightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{U}}_{p}(R))$;

Rule-D2a: $\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(\sigma_{\mathcal{C}}(R)) \overset{*}{\Rightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{D}}_{p,\mathcal{D}\cup\mathcal{C},\delta}(R))$;

Rule-D2b: $\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(\sigma_{\mathcal{C}}(R)) \overset{\sim}{\Rightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta/\sigma^{\mathsf{ss}}}(R))$;

Rule-D2c: $\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(\sigma_{\mathcal{C}}(R)) \overset{\sim}{\Rightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(R))$;

Rule-V2: $\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(\sigma_{\mathcal{C}}(R)) \overset{\sim}{\Rightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R))$, *if* $|\mathcal{D} \cap \mathcal{C}| \ll |\mathcal{D} \cup \mathcal{C}|$.

The selectivity of $\sigma_{\mathcal{C}}$ on R, denoted as σ^{ss}, is $|\sigma_{\mathcal{C}}(R)|/|R|$.

Finally, we list the rules for pushing samplers past joins.

Proposition 9 (Pushing past Join)**.** *For relations R_1 and R_2, with columns $\mathcal{C}_i$ respectively and an equi-join $\bowtie_{\mathcal{C}}$ on columns $\mathcal{C}$, we have*

Rule-U3: $\Gamma^{\mathsf{U}}_{p}(R_1 \bowtie_{\mathcal{C}} R_2) \overset{\mathsf{c}}{\Rightarrow} \Gamma^{\mathsf{U}}_{p_1}(R_1) \bowtie_{\mathcal{C}} \Gamma^{\mathsf{U}}_{p_2}(R_2)$, *if* $p = p_1 \cdot p_2$;

Rule-D3a: $\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(R_1 \bowtie_{\mathcal{C}} R_2) \overset{*}{\Rightarrow} \Gamma^{\mathsf{D}}_{p,\mathcal{D}\cup\mathcal{C},\delta}(R_1) \bowtie_{\mathcal{C}} R_2$;

Rule-D3b: $\Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(R_1 \bowtie_{\mathcal{C}} R_2) \overset{\sim}{\Rightarrow} \Gamma^{\mathsf{D}}_{p,\mathcal{D},\delta}(R_1) \bowtie_{\mathcal{C}} R_2$, *if* $\mathcal{D} \subseteq \mathcal{C}_1$;

Rule-V3a: $\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R_1 \bowtie_{\mathcal{C}} R_2) \overset{*}{\Rightarrow} \Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R_1) \bowtie_{\mathcal{C}} \Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R_2)$, *if* $\mathcal{C} = \mathcal{D}$ *or* $\mathcal{D} \subseteq \mathcal{C}_1, \mathcal{C}_2$;

Rule-V3b: $\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R_1 \bowtie_{\mathcal{C}} R_2) \overset{\sim}{\Rightarrow} \Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R_1) \bowtie_{\mathcal{C}} R_2$, *if* $\mathcal{D} \subseteq \mathcal{C}_1$.

Note that Rule-U3 includes the cases when uniform sampler is pushed to only one side, i.e., set $p_1 = 1$ or $p_2 = 1$. Observe that for rules D3a, D3b and V3b, wherein the sampler is only pushed to R_1, there are analogous rules that push the sampler only to R_2.

C. GENERAL AGGREGATIONS

So far, we have considered groups with SUM-like aggregates. Here, we extend our analysis to other aggregations such as `COUNT` and to the case where a result has multiple aggregations such as `SELECT x, SUM(y), COUNT(z)`. Quickr allows users to annotate user-defined aggregates with functional expressions which it uses to obtain various accuracy measures; details are left for future work.

Other aggregations: Analyzing `COUNT` directly follows from `SUM` by setting $w(t) = 1\ \forall t$. `AVG` translates to $\frac{\texttt{SUM}}{\texttt{COUNT}}$ but its variance is harder to analyze due to the division [24]. In implementation, Quickr substitutes `AVG` by `SUM/COUNT` and divides the corresponding estimators. Quickr also supports `DISTINCT`, which translates to a group with no aggregations and `COUNT(DISTINCT)`. Error for the former is akin to missing groups analysis. For `COUNT(DISTINCT)`, the value estimator varies with the sampler– by default it is the value computed over samples. Only when universe sampling is on the same columns, the value over samples is weighted up by the probability. Further, distinct sampler offers unbiased error and zero variance. With uniform sampler, the variance is small but there is some negative bias since though unlikely due to Quickr's requirement for sufficient support, some rare values may not appear. We defer analyzing the error for other aggregations to future work.

Multiple aggregation ops: Quickr naturally extends to the case when multiple aggregations are computed over the same sampled input relation. The key observation is that the estimators of true values for each aggregation only require the value in the sampled tuple, the corresponding *weight* which describes the probability with which the tuple was passed and in rare cases the type of the sample (e.g., for `COUNT DISTINCT`). The first two are available as columns in the sampled relation. The third we implement as a corrective rewriting after `ASALQA` chooses the samplers.

D. ADDITIONAL RELATED WORK

We mention a few relevant threads of substantial research: synopses and sketches [15, 20, 42], view matching [28] and data streams [13, 31, 34]. Sketches and synopses on input datasets can offer more speed-up than samples. However, they are generally harder to reason with for ad-hoc queries and hence Quickr prefers sampling. However, we do use the heavy-hitter sketch to reduce memory needs of our distinct sampler. Sampling in data streams has some similar constraints to what we pose on our samplers such as bounded memory footprint and streaming execution. However, while Quickr samples partitioned data in parallel for ad-hoc queries, stream management systems focus on (known) queries that effectively execute forever. Reusing sampled sub-expressions generated by previous queries can improve Quickr and hence view matching is an area of future work for us.