Experiences with Approximating Queries in Microsoft's Production Big-Data Clusters

Srikanth Kandula, Kukjin Lee, Surajit Chaudhuri, Marc Friedman
Microsoft
{Srikanth, KuLee, SurajitC, MarcFr}@microsoft.com

ABSTRACT

With the rapidly growing volume of data, it is more attractive than ever to leverage approximations to answer analytic queries. Sampling is a powerful technique which has been studied extensively from the point of view of facilitating approximation. Yet, there has been no large-scale study of effectiveness of sampling techniques in big data systems. In this paper, we describe an in-depth study of the sampling-based approximation techniques that we have deployed in Microsoft's big data clusters. We explain the choices we made to implement approximation, identify the usage cases, and study detailed data that sheds insight on the usefulness of doing sampling based approximation.

PVLDB Reference Format:
Srikanth Kandula, Kukjin Lee, Surajit Chaudhuri, Marc Friedman. Experiences with Approximating Queries in Microsoft's Production Big-Data Clusters. *PVLDB*, 12(12): 2131-2142, 2019.
DOI: https://doi.org/10.14778/3352063.3352130

1. INTRODUCTION

Executing complex data analytics queries on ever increasing datasets costs time and money. For example, breaking down the latency of Bing searches into amounts that are incurred at each of the components involved in answering the search by joining logs collected at various servers and then grouping the results on query features, today, requires many thousands of compute hours for an hour's worth of searches at Bing.

Approximate query execution, e.g., running the query on a sample of the input, can lower the latency and cost of running complex queries on large datasets. This point is not lost on the application developers who routinely use approximations ranging from algorithmic techniques to hardware primitives to execute their query on a smaller instance of the data. However, despite substantial research [16, 24, 31, 34, 35, 39, 40] going beyond application-level approximations to supporting approximation as a built-in functionality in data platforms has proven elusive.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352130

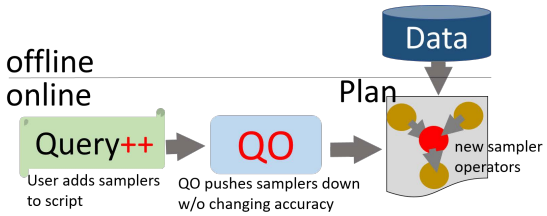


Figure 1: Workflow for Query-time Sampling

We believe that this is because currently known techniques do not satisfy the following prerequisite: for a large subset of the queries on a data platform, the approximation technique should offer sizable savings (cost or latency) with only a small effect on the answer quality and the technique should have a user-understandable error model and a small overhead. See [21] for a detailed reasoning on why currently known techniques fall short in achieving this prerequisite. Techniques that use previously computed samples of the input [16, 18, 22, 40] cannot handle complex queries satisfactorily and have sizable space overheads to store different kinds of samples [34]. Online aggregation techniques [30, 31, 32] require input to be ordered randomly and require complex changes to relational operators (e.g., for join, need ripple join [30] or SMS join [32]).

We have implemented support for query time sampling (see Figure 1) in production big-data clusters at Microsoft [20]; these clusters consist of tens of thousands of multi-core, multi-disk servers and are used by developers from many different businesses including Bing, Azure and Windows. In total, the clusters store a few exabytes of data and are primarily responsible for all of the batch analytics at Microsoft. We also have a simplified version available publicly as part of the Azure Data Lake Analytics [8, 9].

In the query time sampling paradigm, the platform cedes control of accuracy to the application developer; i.e., we extend our query languages with sampling operators and the user expresses query-specific accuracy needs by using appropriate samplers in the query expression. Sampling operators have been discussed for a few decades now [39]; however, most prior works focus exclusively on uniform samples [19, 27, 26, 28]. In contrast, we find that large query coverage requires new sampler operators that can be used below group-by and join; the uniform sampler is not suitable for such use because it can miss small groups when used below group-by and when used on multiple join inputs either reduces data reduction or leads to high error. In addition to the uniform sampler, we support two new sampling opera-

```
SELECT n_name,
   SUM(l_extendedprice*(1-l_discount)), COUNT(*)

FROM lineitem
JOIN orders ON l_orderkey = o_orderkey
JOIN supplier ON l_suppkey = s_suppkey
JOIN nation ON s_nationkey = n_nationkey
JOIN region ON n_regionkey = r_regionkey
WHERE r_name=':1' ∧ o_orderdate ∈ [':2',':3')
GROUP BY n_name
```

(a) Original query

```
SELECT n_name,
  SUM(l_extendedprice*(1-l_discount)*w), SUM(w)
FROM SAMPLE (
  SELECT * FROM lineitem
  JOIN orders ON l_orderkey = o_orderkey
  JOIN supplier ON l_suppkey = s_suppkey
  JOIN nation ON s_nationkey = n_nationkey
  JOIN region ON n_regionkey = r_regionkey
  WHERE r_name=':1' ∧ o_orderdate ∈ [':2',':3')
) UNIFORM (0.05) WITH WEIGHT AS w
GROUP BY n_name
```

(b) Sampled query

Figure 2: Illustrating the use of samplers for TPC-H query #5; here, the user expresses the intent to compute the group-by and aggregation over a 5% uniform sample of the joined relation. Our QO pushes sampler down automatically w/o changing accuracy; see Figure 3.

tors, *distinct* and *universe*, which can respectively be used before a group-by and a join [34].

Using these sample operators, queries will see a sizable improvement in cost and latency only if substantial work executes in the query plan after the samplers; that is, samplers execute early in the query plan and subsequent operators benefit from working on smaller sampled relations. Hence, query optimizer transformations which push samplers down without affecting plan accuracy can improve performance. Consider the example in Figure 2 which is query #5 from the TPC-H workload. The actual query is on the left, the user-specified query is on the right and, the plan executed after QO transformations is shown in Figure 3; here, the user indicates that a 5% uniform sampler suffices and the QO has pushed that sampler past the various key foreign-key joins onto the `lineitem` fact table. Doing so speeds-up the query because the joins execute with a smaller input while the accuracy remains as requested by the user[1].

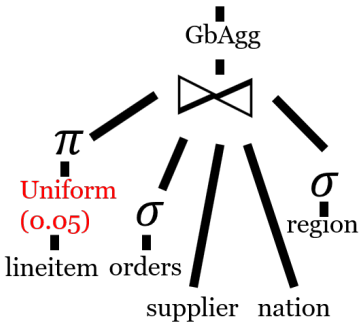


Figure 3: Plan after sampler pushdown

In this paper, we analyze many tens of thousands of production queries that use approximations and describe our experiences engaging with application developers. We measure the prevalence of usage of samplers, comment on the use-cases, breakdown the usage by sampler type and choice of sampling parameters. We also report micro-benchmarks on the sampler operators (throughput, memory usage); the usefulness of the query optimization (how much do samplers get pushed down?); as well as macro results of the value from using query-time sampling; that is, the cost, latency and output accuracy of sampled plans compared to the corresponding unsampled query plan. To the best of our knowledge, this is the first study to present a large-scale and detailed analysis of production queries that use approximations. Our key findings are:

- All three sampler operators are roughly equally used pointing at the value of supporting all of them.
- Highly efficient sampler implementations with minimal requirements (e.g., sublinear memory footprint, one pass implementation with no requirements on input partitioning or sorted-ness) is crucial; otherwise, there would be sharp reduction in gains.
- The use-cases in production are substantially more complex than the above example; large groups, highly parallel plans and complex or user-defined aggregates, selections, projections and joins are common. A sizable fraction of the datasets are unstructured lacking schema information such as keys etc. QO pushdown rules for samplers had to be carefully constructed to work well on complex queries.
- In addition to using approximations to speed-up aggregation queries, we see our sampler operators being used in a few different ways: (1) to construct training data for machine learning applications by slicing relations on some feature value (e.g., positive and negative examples) and sampling each slice differently; not doing so affects model quality because the model may become blind to the less common cases; (2) *explicit sampling* where the developer's intent is not to generate an approximate version of the unsampled query but rather intentionally chooses to run the query on a sample and (3) *output sampling*, i.e., using the sample operator as the "last operator" to produce a smaller copy of some query output that can be investigated more carefully by humans or consumed by other tools. To our knowledge, these use-cases have not been reported before. We have seen uniform, distinct and universe samplers used in all of these use cases.
- Data scientists are comfortable reasoning about samplers with minimal training; however, with a majority

[1]The query on the right also shows how user's rewrite their aggregates; the rewrites shown are unbiased estimators of the true answer, that is, the expected error over many different query executions is zero. Confidence intervals (based on variance estimators and the CLT [23]) can be computed as additional aggregations [34].

of cluster users, we observe substantial cognitive overhead in terms of choosing which sampler to use and reasoning about confidence intervals. Both of these issues hamper adoption.

- Another barrier to adoption is the lack of an accuracy guarantee; even if the query does well during testing, there is no guarantee that it will work well on unseen data. This concern is less pronounced for recurring jobs wherein the same query periodically executes on newly arriving data which has been shown to have similar data statistics as previously-seen data [15].

- Some users hesitate to use *any* approximation; even though cost reduces and latency improves, they are unwilling to accommodate even a hint of inaccuracy. This is despite the fact that most log analytics are implicitly sampled results because even a query that processes all log entries only obtains a measurement estimate of how the underlying system is behaving; arguing in this way has helped in some cases.

- Sampling operators can be an extremely powerful tool for experts; on TPC-H [12], we will show sizable reductions in query cost and latency with only a small change in answers (§4.1). Note that this result has zero overhead; that is, no apriori samples [16, 40] or indices [24, 35] need to be maintained, no other relational operators have to change [30, 32] and there is no constraint on how input is partitioned or stored [30, 31, 32].

The primary contribution of this paper is our reported experiences from production use of samplers in big-data clusters at Microsoft. These clusters are used by thousands of developers and we present results from over six months of usage. Additional technical contributions include:

- Although the deployed samplers are like previously published descriptions [34], there are some key differences and we highlight these in §2.1.

- The plan transformation rules which pushdown samplers have been substantially simplified and extended relative to prior work [34]; in large part, the simplification is possible because script owners specify the sampler to use whereas in prior work, the QO had to choose the most appropriate sampled plan [34]. We discuss the deployed transformation rules in §2.2.

2. QUERY-TIME SAMPLING

As shown in Figure 1, the two main steps in query-time sampling are (1) users add sample operators to their scripts and (2) the query optimizer uses a collection of transformation rules to push down the samplers while ensuring that accuracy remains the same. In this section, for each of these steps, we discuss design details and measurements from the use of samplers in production at Microsoft's big-data clusters. Note that the results here are from an analysis of production logs over the period of July 2018 to Feb 2019.

2.1 Sampler operators

Our production system exposes three types of samplers.

Table 1: Breaking down sampler types and the prevalence of weight column.

Types of samplers used	UNIFORM: 36.3% DISTINCT: 23.5% UNIVERSE: 40.2%
Weight column requested	4.31% of samplers

Table 2: The number of samplers used in each query.

Number of samplers per query	1	2	3	> 3
In query (logical)	67%	14%	19%	0.7%
In query plan (physical)	66%	14%	19%	1.7%

`Uniform (`p`)` picks incoming rows uniformly at random with the specified probability p. The size of the output is governed by a binomial distribution and the expected size is a p fraction of the input. Our implementation of this sampler has no appreciable memory footprint; it is zero-copy, and does not access the row memory; all of these aspects lead to a pipelinable operator with very high throughput.

`Universe (`p`, `$\mathcal{D}$`)` picks a p fraction of the values of the columns in set $\mathcal{D}$; all rows with such value are passed by this sampler. This sampler is useful because it can be pushed down to multiple inputs of a join; for example, joining the universe samples of two relations is identical to taking the same universe sample of the join result [34]. The universe sampler also has no appreciable memory footprint but it accesses row memory to retrieve the values of columns in set $\mathcal{D}$ and computes strong hash functions over these values. Hence, the throughput of the universe sampler is somewhat less than that of the uniform sampler above.

`Distinct (`p`, `$\mathcal{D}$`, `f`)` passes at least f rows per distinct value of the columnset $\mathcal{D}$ and the other rows are passed with at least p probability. The distinct sampler can be used below a group-by by choosing $\mathcal{D}$ to be the group columns because every unique value of $\mathcal{D}$ will be represented in the sample unlike the case of a uniform sample which can miss small groups. To distinguish between rows that are passed for the frequency check and the probability check, this sampler emits a weight column for each row which is either 1 or $\frac{1}{p}$. The distinct sampler has internal state; it uses a heavy hitter sketch [36] to track the more frequent distinct values of columnset $\mathcal{D}$ as described in [34]. The memory footprint is at most logarithmic in the number of input rows (see [36] for proof) but we observe that it is much smaller in practice. This sampler has smaller throughput than the two samplers above.

2.1.1 Samplers in production clusters

Sampler usage by type: Table 1 breaks-down the usage of these three samplers over all the production jobs that used any sampler during the examined period in all of our big-data clusters. We see that all three samplers are used roughly evenly. Note this finding contrasts with results on the TPC-DS benchmark, presented in [34], where the uniform and distinct samplers were used more often.

Table 2 shows that roughly one third of the queries in the examined cluster use multiple sampler operators; the largest number of sampler operators observed in a query expression

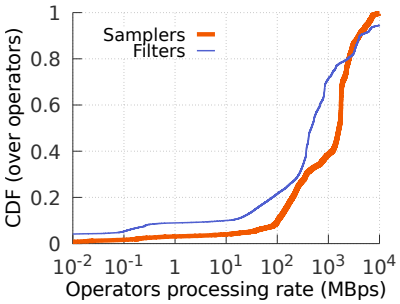


Figure 4: Data processing rate of the sampler operators (elapsed time attributed to sampler divided by the input size). We show the processing rate of filters for comparison.

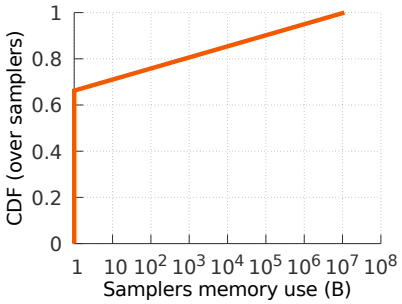


Figure 5: Memory used by sampler operators.

(which we term logical operators) is 8. When the query optimizer pushes samplers below a join it can, in some cases, push samplers to multiple join inputs. Hence, as shown in Table 2, there can be more samplers in the plan output by the QO than the samplers specified in the query expression; the largest number of such physical sampler operators observed was 92.

Sampler throughput: Figure 4 shows a CDF of the processing rate of all of the sample operators that executed during the examined period in our clusters. The processing rate metric is computed by dividing the input size processed by an operator with the time attributed to that operator. The servers are Xeon class with 32 cores, 128GB RAM, multiple SSD drives, 10Gbps network interfaces and the interconnect has nearly the full bisection bandwidth [29]. In the SCOPE execution runtime, each *task* has an internal graph of operators and task execution proceeds by the output operator(s) pulling rows from their upstream operators until all of the tasks' input data has been processed. Operators can maintain local state and rows are passed between operators as memory pointers. We see that the median processing rate for samplers is over 1000MBps; over 95% of the samplers process input at over 100MBps; most of the operators having low processing rate process very little data and their processing rate is low because of constant overheads. The uniform and universe sampler operators as noted above have higher processing rates than the distinct sampler. The figure also shows a CDF of the processing rate for filter (predicate) operators with a thin blue line; filters are one of the most efficient operators in a relational platform because they require no local state and can be implemented with zero copy. From the figure, we see that overall the sampler operators have a processing rate similar to that of the filter operators. We describe some of the optimizations that were necessary to yield such highly performant samplers shortly.

Sampler's memory footprint: Figure 5 depicts the memory used by the samplers; our operator runtime tracks memory used by each operator. The figure plots the maximum memory sized used during the operator's lifetime. Note that the x axes is in log scale. The figure shows that a significant majority of the samplers have no appreciable memory footprint; the others, which are all distinct samplers, use about 10MB. This substantiates our earlier assertion that the memory footprint of the hashed heavy hitter sketch is small in practice.

Performant operator implementations: Implementing uniform and universe samplers in a highly performant manner is relatively straightforward. We take care to not access the row in memory (in the case of the uniform sampler) and to only access the necessary columns (for the universe sampler). To generate "high quality" random numbers quickly we use the `std::mersenne_twister_engine` which has been shown to outperform other widely available random number generators (RNGs) [4]; "better" RNGs use processor-specific instructions and do not easily generalize across architectures [4].

The distinct sampler has a more complex implementation. One key requirement is to track groups [2] that have large numbers of rows; recall that the distinct sampler will output at least a specified number of rows for each group and so only groups having more rows than the specified number will be randomly sampled. As noted above, our implementation only tracks groups that are heavy hitters to reduce the memory footprint; however, note that doing so will cut into sampling gains because groups that have more rows than the specified number but are not heavy hitters will needlessly have more rows pass through this implementation. Figure 6 plots the gap between the sampler's specified probability and the actual operator selectivity; note that the gap is small in general [3]. A second challenge is that the heavy hitter sketch that we use [36] makes frequent insertions and deletions into a dictionary; a naive implementation, e.g., using `std::map` can fragment memory leading to a much larger footprint than the actual number of stored elements and so we built a customized dictionary that works over a memory pool that we manage. A third challenge is from supporting parallel executions of the distinct sampler. Note that the uniform and universe samplers are trivially parallelizable but not the distinct sampler. Consider the case of a group with $10f$ rows that is processed by 11 tasks in parallel with each task having a distinct sampler with a frequency cut-off of f; if the group's rows are uniformly distributed across the samplers all $10f$ rows will pass because no individual sampler sees more than f rows. Avoiding this requires the input of the distinct sampler to be partitioned on the distinct columnset which may require a network shuffle; shuffles are expensive in parallel processing because they lead to a scheduling barrier and/or writes to stores and data movement across the network. Our implementation does not introduce this shuffle

[2]We colloquially use *group* to refer to the set of rows that have the same value in the columnset of the distinct sampler.
[3]Even a perfectly implemented distinct sampler will have a non-zero gap because it passes all rows for small groups.

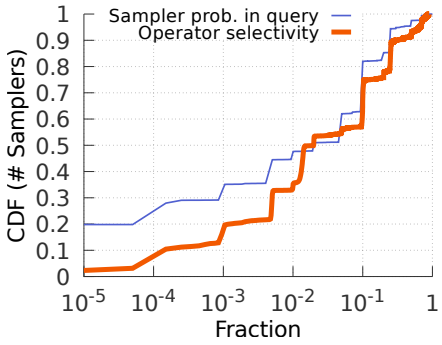


Figure 6: Desired sample probability vs. actual selectivity.

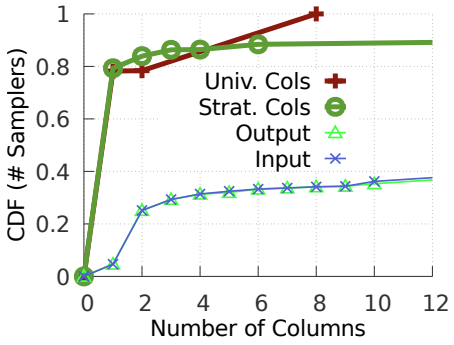


Figure 7: Number of columns in the input columnset for the distinct sampler ("Strat. Cols") and the universe sampler ("Univ. Cols"). The figure also shows the size of the input and output relations of sampler operators.

to retain high performance but, as we already saw in Figure 6, the cost of doing so in terms of passing more rows than needed is small in our experience. We will note an exception in a case study, later, in §4.2.3.

Characterizing sampler parameters: Figure 6 shows, as a CDF over all samplers, the sampling probability parameter that the authors specify as input as well as the actual selectivity achieved by the operator. Note that the x-axes is in log scale. About 30% of the samplers pick less than 0.001 or 10^{-3} fraction of the input; these are commonly jobs with very large inputs. Many samplers also pick 10% (or 10^{-1} fraction) of the input; this was the recommended setting in our manual; we based this recommendation on the largest probability parameter that can achieve a sizable speed-up for jobs. Fewer than 10% of the samplers use a probability assignment above 0.25. The figure also shows that the actual selectivity observed is, in some cases, larger than the desired probability value; this is almost entirely because of the distinct samplers for reasons noted in the preceding paragraph.

Figure 7 shows the set sizes of various relevant column sets. We see that 80% of the universe sampler instances are over a single column; the largest is 8 different columns. Typically, the universe columns are the columns used in an equijoin condition because when a universe sampler is pushed to multiple inputs of join, the universe columnset has to match the columns in the corresponding equijoin condition.

Figure 7 also shows that 90% of the input columnsets for the distinct sampler have between 1 and 6 columns (the "Strat. Cols" line); 1 column is most likely and we see a roughly uniform distribution between 2 to 6 columns. When printing plans, our implementation truncates column-sets that are larger than 10 columns; roughly 10% of the distinct samplers stratify on more than 10 columns. The large size is because stratifying on the more distinctive columns in groups as well as columns that appear in highly selective predicates is necessary to be able to push distinct sampler below group-by and selections respectively. The large size of the stratification column set and the number of different column sets that are used (not shown in this figure) indicate that apriori generation of stratified samples [16, 22] may be less useful; such methods maintain many different stratified samples, one per stratification column set, and can consume a lot of space. Moreover, these samples have to be refreshed as datasets evolve; query-time sampling does not have such maintenance overheads.

Finally, Figure 7 shows the sizes of the input and output relations of the samplers; roughly 60% of the samplers apply on relations having over 10 columns (recall our plan print logic only prints the first 10 columns). The figure shows that output relations are roughly as large as the input relations likely because the cases when samplers add the weight column (output relation is larger) are offset by cases where samplers fuse with projections and drop redundant columns.

Deterministic re-execution: Distributed job schedulers can execute a task multiple times during failure scenarios. It is preferable that a task yields the same output when presented with the same input rowset. We achieve this by seeding the samplers in each task; the seeds are code-generated by the QO at plan compilation. Seeding also ensures that re-running a job can produce the same output. Using a different seed, as expected, will randomize the output[4].

Hash collisions affect the distinct sampler in a specific manner which is worth noting. The heavy hitter sketch [36] used by the distinct sampler only records hash values of the stratification columnset and so hash collisions can lead to some groups receiving fewer rows from the sampler than the desired number of rows. We make the practical likelihood of this case small by using "high quality" hash functions and 64 bit hash values [17, 5]. The likelihood of delivering fewer than the desired minimum number of rows or of missing a group entirely depends on the number of groups, the number of heavy hitters and the number of rows contained in a group; only tracking the frequency of a small number of heavy hitters as we do by using a heavy hitter sketch reduces the likelihood of hash collisions.

2.1.2 *Language extension for samplers*

We extend the SCOPE language [20] which mashes relational and imperative aspects; a query in SCOPE is an imperative list of named SQL-like statements. Users can specify samplers in their query by either adding a new sample statement (as shown in Figure 2b) or adding a sample clause to an existing SQL select statement. While samplers-as-a-clause is easier to insert into a script, clauses have some specific constraints including their inability to change the

[4]Note that with some job schedulers the input of a task can change between its original execution and re-execution (e.g., with work-stealing schedulers [37]); in such cases seeding is not sufficient to ensure deterministic output from samplers and a more nuanced implementation is needed.

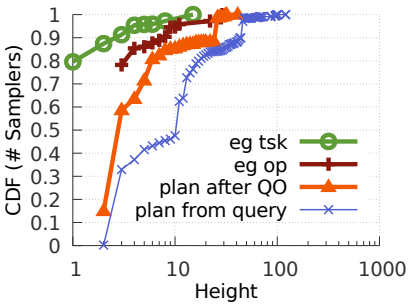


Figure 8: Locations of the samplers in the plan as well as in the task execution graph.

output schema which is needed when the sampler outputs a weight column. Recall that select statements have multiple clauses such as `HAVING`, `WHERE`, `GROUP BY` etc.; the precedence order among these clauses is such that the sample clause executes after the `WHERE` clause and before the `GROUP BY` clause [6]. The syntax for the clause is:

Sample_Clause :=
  Query_Expression `SAMPLE UNIFORM ( row_fraction )`.

An example usage of sample-clause is:

`SELECT * FROM` input `SAMPLE UNIFORM` (0.1)

A sample statement takes as input a table valued function (TVF) and returns a TVF. The syntax is:

Sample_Statement :=
  `SAMPLE` Rowset Sampler_Details [`WITH WEIGHT AS` Ident]

Sampler_Details :=
  `UNIFORM ( row_fraction )` |
  `ON` Ident_List `UNIVERSE ( row_fract )` |
  `ON` Ident_List `DISTINCT ( row_fract , min_row_cnt )`

Some examples follow:

`SAMPLE` @input `UNIFORM` (0.1)

`SAMPLE` @input `ON` c_1, c_2 `DISTINCT` $(0.1, 3)$ `WITH WEIGHT AS` w

`SAMPLE` @input `ON` y `UNIVERSE` (0.1)

2.2 Sampler pushdown rules

We use plan transformation rules to push samplers down without affecting accuracy. In general, doing so improves plan performance because more relational operators execute on the reduced size relations obtained after sampling. We use a strict notion of preserving accuracy: two query expressions are said to be equivalent if every subset of rows in the output relation has the same probability of occurring in either expression. We use this property because when such accuracy-preserving plan transformation rules are applied, the unbiased estimators of downstream aggregates and the estimators of aggregate variance (for confidence intervals) remain unmodified. Furthermore, we eschew using some of the transformation rules proposed by prior work [33, 34] which preserve accuracy only when the underlying relations have certain data statistics; e.g., all groups have a sufficiently large number of rows. We do not use these rules in production because due to the complexity of queries and the sheer volume of datasets, the cluster does not have metadata information required to verify that these data statistics hold. Consequently, the transformation rules that we use are independent of data statistics.

We describe the effects of sampler pushdown rules on the plans of production queries before discussing more details of the rules that we have implemented.

Characterizing value of sampler pushdown: Figure 8 shows the CDFs of four different *heights* of all the observed sampler operators. Starting from the bottom right, the *plan after QO* and *plan from query* curves correspond to the heights of the sampler operator in the query optimizer's output plan and the literal plan from the query respectively. Operators in the actual execution graph do not have a one-to-one match with operators in the plan because multiple plan ops can be fused into the same runtime op and conversely, due to parallel execution, shuffles, sorts or partitions may have to be added [20]. Hence, the *eg op* curve denotes the height of the samplers in the operator graph. Finally, the *eg tsk* curve shows the height of the task, in the execution graph, that the sampler belongs to; recall that each task executes an internal operator graph. Our *height* metric is 1 for the leaves, e.g., the operator that reads input or the task that reads input, and increases by 1 per operator or task as the case may be.

We note several points from the results in Figure 8. First, the height of samplers in the plan output by the query optimizer (*plan after QO* curve) is smaller than the samplers height in the plan specified by the user (*plan from query* curve). This directly corresponds to samplers being pushed down. We see that the gap ranges from 1 to well over 10; note that the x axes is in log scale. Next, from the *eg tsk* curve, we see that 90% of the samplers execute in the first pass on data (task height = 1); over 98% of samplers execute in the first three passes of data (task height ≤ 3); the latest sampler executed in the 11'th pass on data. Third, we see that the plans are complex; the operator height of the sampler in the actual execution graph (*eg op* curve) is often much larger than its task height (*eg tsk* curve); this is because tasks execute multiple operators. In summary, we see evidence that plan transformation rules are effective, leading to smaller heights for samplers in the output plans. We also see that most samplers (but not all) execute in the first few passes over data potentially speeding-up all operators that execute in subsequent tasks.

We next describe the sampler pushdown rules that we have implemented.

Sampler below projection: Consider a relation R and a projection π which renames the columns in set $\mathcal{C}_a$ with the corresponding columns in set $\mathcal{C}_b$ and generates a new column c using as input columns in the set $\mathcal{C}_c$. Note this implies that c is functionally dependent on $\mathcal{C}_c$, i.e., $\mathcal{C}_c \vdash c$. Extending to cases with multiple functions is straightforward. Rules U1, V1 and D1 in Table 3 show the transformation rules that push samplers below such a projection and the conditions required to apply these rules. These transformations encode the previously specified equivalence; that is, the likelihood of any set of rows appearing in either expressions is identical. In V1 and D1, the universe and stratification columns are renamed; moreover the rules apply only if the newly generated column c is not used by the sampler (e.g., in V1 and D1) or if c is functionally dependent on columns that are already present in the stratification set (e.g., in rule D1).

Table 3: Plan transformation rules. See §2.2. Here, $\Gamma^{\mathsf{U}}_{p}, \Gamma^{\mathsf{V}}_{p,\mathcal{D}}$, and $\Gamma^{\mathsf{D}}_{p,\mathcal{D},f}$ are a uniform, universe and distinct sampler respectively with probability p, universe and stratification columns from set $\mathcal{D}$ and a frequency target of f. The notation $\mathcal{D}_{\mathcal{C}_b \to \mathcal{C}_a}$ denotes replacing columns in $\mathcal{D}$ that are in the set $\mathcal{C}_b$ with corresponding columns from the set $\mathcal{C}_a$. For relation R, we use R_c to denote the columns in R. When relations R and S are joined, we use $\mathcal{D}_{S \to R}$ to denote a set generated by replacing the columns in set $\mathcal{D}$ that belong to S_c with equivalent columns in R_c as per the equijoin conditions, e.g., if $\mathcal{D} = \{r_1, s_1, s_2\}$ and the join condition is $r_3 = s_1$ where r_i and s_i are from the relations R and S, then $\mathcal{D}_{S \to R} = \{r_1, r_3, s_2\}$.

	Transformation	Condition
Rule-U1	$\Gamma^{\mathsf{U}}_{p}(\pi(R)) \overset{*}{\Leftrightarrow} \pi(\Gamma^{\mathsf{U}}_{p}(R))$	–
Rule-V1	$\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(\pi(R)) \overset{*}{\Leftrightarrow} \pi(\Gamma^{\mathsf{V}}_{p,\mathcal{D}_{\mathcal{C}_b \to \mathcal{C}_a}}(R))$	if $c \notin \mathcal{D}$
Rule-D1	$\Gamma^{\mathsf{D}}_{p,\mathcal{D},f}(\pi(R)) \overset{*}{\Leftrightarrow} \pi(\Gamma^{\mathsf{D}}_{p,\mathcal{D}_{\mathcal{C}_b \to \mathcal{C}_a},f}(R))$	if $c \notin \mathcal{D}$ or $\mathcal{C}_c \subseteq \mathcal{D}$
Rule-U2	$\Gamma^{\mathsf{U}}_{p}(\sigma_{\mathcal{C}}(R)) \overset{*}{\Leftrightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{U}}_{p}(R))$	–
Rule-V2	$\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(\sigma_{\mathcal{C}}(R)) \overset{*}{\Leftrightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R))$	–
Rule-D2	$\Gamma^{\mathsf{D}}_{p,\mathcal{D},f}(\sigma_{\mathcal{C}}(R)) \overset{*}{\Leftrightarrow} \sigma_{\mathcal{C}}(\Gamma^{\mathsf{D}}_{p,\mathcal{D},f}(R))$	if $\mathcal{C} \subseteq \mathcal{D}$
Rule-U3	$\Gamma^{\mathsf{U}}_{p}(R \bowtie_{\mathcal{C}} S) \overset{*}{\Leftrightarrow} \Gamma^{\mathsf{U}}_{p}(R) \bowtie_{\mathcal{C}} S$	if $\mathcal{C}$ is a primary-key in S
Rule-V3a	$\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R \bowtie_{\mathcal{C}} S) \overset{*}{\Leftrightarrow} \Gamma^{\mathsf{V}}_{p,\mathcal{D}_{S \to R}}(R) \bowtie_{\mathcal{C}} S$	if $\mathcal{D}_{S \to R} \subseteq R_c$ and $\mathcal{C}$ is a primary-key in S
Rule-V3b	$\Gamma^{\mathsf{V}}_{p,\mathcal{D}}(R \bowtie_{\mathcal{C}} S) \overset{*}{\Leftrightarrow} \Gamma^{\mathsf{V}}_{p,\mathcal{C}}(R) \bowtie_{\mathcal{C}} \Gamma^{\mathsf{V}}_{p,\mathcal{C}}(S)$	if $\mathcal{C} = \mathcal{D}$
Rule-D3	$\Gamma^{\mathsf{D}}_{p,\mathcal{D},f}(R \bowtie_{\mathcal{C}} S) \overset{*}{\Leftrightarrow} \Gamma^{\mathsf{D}}_{p,\mathcal{D}_{S \to R},f}(R) \bowtie_{\mathcal{C}} S$	if $\mathcal{C} \subseteq \mathcal{D}_{S \to R} \subseteq R_c$ and $\mathcal{C}$ is a primary-key in S.

Sampler below selection: Consider a relation R and let $\sigma_{\mathcal{C}}$ denote a selection over columns in set $\mathcal{C}$. Rules U2, V2 and D2 in Table 3 show transformation rules that push samplers below selection. These rules also encode equivalence. Rule D2 relies on functional dependence between the stratification columns and the predicate columns. For complex predicates these rules can be applied to each clause in a CNF of the actual predicate; we use standard QO transformations to explore the space of predicate rewriting.

Sampler below join: Given two relations R and S being equijoined on columns $\mathcal{C}$, rules U3, V3 and D3 in Table 3 show how samplers push below joins. As above, all specified rules encode equivalence.

Since all of the sampler pushdown rules shown in Table 3 encode equivalence, the *accuracy* of aggregates computed over these expressions is identical. Moreover, the expressions on the right are less expensive to evaluate in most cases (but not always, e.g., in V2 and D2, the sampler may be costlier to evaluate than a simple predicate). We have implemented all of these transformations as plan substitutions to simplify implementation in the QO; in practice, even if a particular pushdown does not improve performance, it can lead to other pushdowns and the cumulative effect of multiple pushdowns often leads to better performing plans because most of the pushdowns improve performance.

3. CHARACTERIZING JOBS THAT USE SAMPLERS

Thus far, we have discussed operator-level measurements (e.g., usage, throughput, memuse, parameters) and plan-level measurements (height of samplers before and after pushdown). In this section, we broaden our scope to examine aspects of the jobs that use sampler operators.

Usage frequency: Figure 9 shows the prevalence of samplers in production; in every examined day over the past several months, hundreds of jobs execute with at least one sample operator. There is a small positive trend, i.e., a small increase with time in the number of jobs that use samplers.

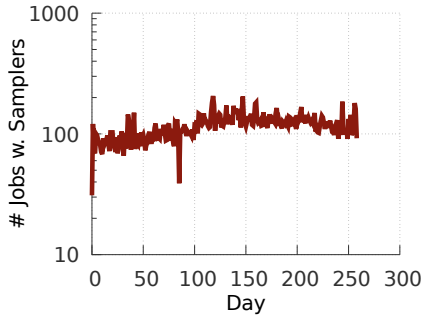


Figure 9: Usage frequency: Hundreds of jobs run every day using sampler operators.

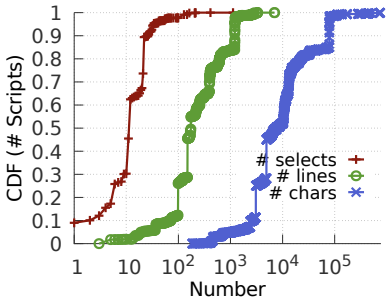
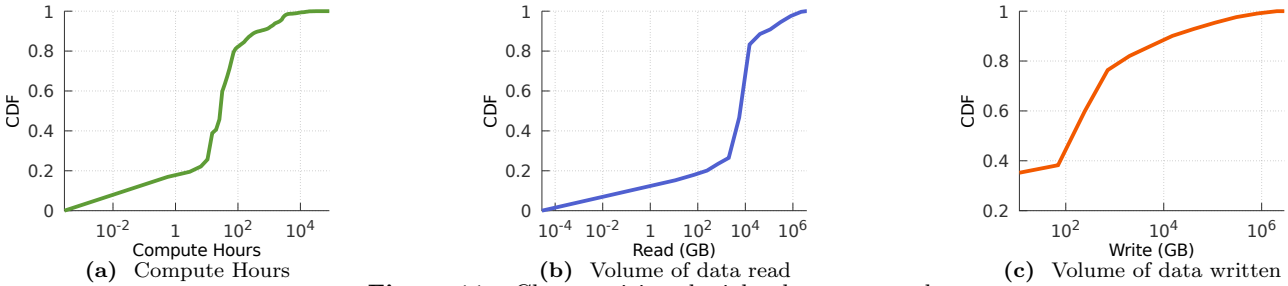


Figure 10: CDF over jobs containing samplers of the number of select statements, lines and characters.

Profiling jobs that use samplers: Figure 10 shows the sizes of the queries that use samplers. We see that the median query has in excess of 10 SQL-like `SELECT` statements, upwards of a few hundreds of lines in all and a few tens of thousands of characters. In contrast, the median query in TPC-DS [13] when expressed in SCOPE has 2 `SELECT` statements. Note also the substantial tail; the x axes is in log-scale and we have observed queries with several hundred `SELECT` statements. In summary, the queries that use samplers are rather complex.

Figure 11 characterizes the jobs that use samplers in other



(a) Compute Hours **(b)** Volume of data read **(c)** Volume of data written

Figure 11: Characterizing the jobs that use samplers

ways; it shows a CDF over the jobs that use samplers, their total compute hours, the total data read and written. We see that at least 50% of the jobs take over 60 hours of execution time (Figure 11a), read over 10TB of data (Figure 11b) and write over 100GB (Figure 11c). A sizable fraction of the jobs are much *larger*. This shows that the usage of samplers is distributed among different types of jobs; small jobs that finish within minutes and massively parallel jobs that take several thousands of cluster hours use samplers.

Understanding use-cases: We identified the following common themes among the queries that use samplers.

- **Computing aggregate queries quickly with little change in answer quality:** This has traditionally been the expected use-case for approximate query processing; it is perhaps most clearly reflected in benchmarks for decision support queries [12, 13]. Queries having a group-by with one or more aggregates computed over a select-project-join expression can execute more quickly over appropriate samples of the input. Some difficulties for this query class include: operators such as unions, nested queries and downstream ops that rely on the aggregate value such as an ORDER BY and TOP or filtering on the aggregate value etc. (e.g., `value` $>$ 0.5 $*$ `average`). Most previously published works only report results for this use-case [16, 34, 40]. This use-case is also prevalent in our production clusters with the primary difference being that the queries are significantly more complex; groups can contain tens of columns and many queries have user-defined filters, aggregations, projections, joins and group-bys.

- **Explicit sampling:** Another typical case is where queries take a sample somewhere in the query and proceed to perform some complex and detailed computation on the sampled relation. These queries never correct for the effect of sampling unlike queries in the above use-case; that is, sampling is used with some other intent besides approximating the result of the unsampled query.

- **Sampling to construct training datasets for ML:** One predominant special-case of explicit sampling is queries that perform some complex log analytics to generate training and test datasets for machine learning purposes. Some queries use sampling to break up the datasets into training datasets (e.g., pick some p fraction as training and the rest as test). More complex usages include dividing the dataset into silos (e.g., based on predicates over some feature values), sampling each silo with a different probability and then union'ing the results. We believe that the user goal here is to construct representative training sets to ensure that the learnt models are not biased towards just the more frequent examples.

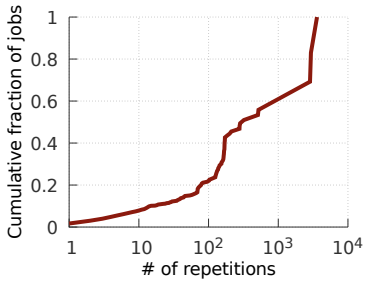


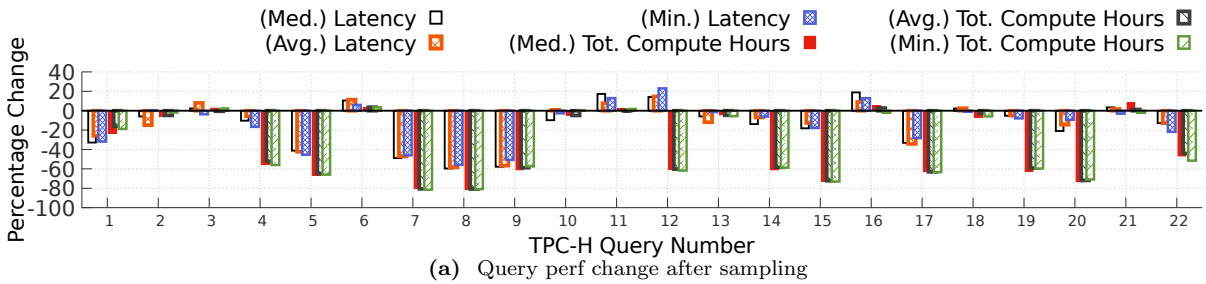
Figure 12: Cumulative fraction of jobs that use samplers as a function of the number of times that job repeats.

- **Output sampling:** In many complex log analytics jobs we see that sampling is used nearly at the output often to generate a small copy of the actual output. The user goal here is most likely to use this smaller copy for manual *judgment* or as input to other queries or external ML tools.

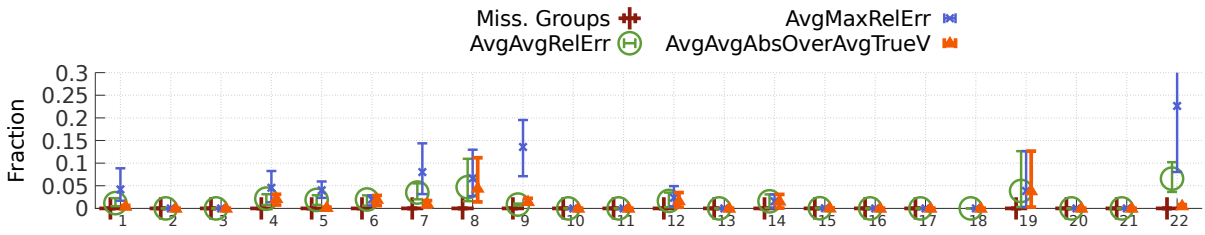
Recurring jobs vs. non-recurring jobs: Figure 12 breaks-down the total number of jobs versus the number of times the jobs repeated during the examined period. Among the jobs that use samplers, only 2% are from non-recurring jobs; that is, jobs that never occur again. Over 80% of the jobs that use samplers repeat at least 100× each (i.e., $y > 0.2 \Rightarrow x \geq 100$). Note, each instance of a job runs the *same query* over different inputs. Typically, jobs run periodically over changing time-windows of logs. As noted earlier, the cognitive overhead that an application developer faces to pick an appropriate sampler (choice of sampler type, parameters such as probability value and columns to use the distinct or universe sample over) amortizes over the multiple repetitions of these jobs because the different inputs of the repeating job instances have similar data statistics [15].

4. PERFORMANCE AND ACCURACY OF JOBS WITH SAMPLERS

Does the use of samplers offer sizable performance improvements relative to the inaccuracy introduced to the answers? We discuss a few specific case studies here beginning with queries from the TPC-H benchmark [12].



(a) Query perf change after sampling



(b) Effect on the query answer

Figure 13: The effects of using sample operators on queries from TPC-H; results are from ten executions of each query on a large shared data-parallel cluster. Input is 100GB.

4.1 TPC-H

We present results on the TPC-H benchmark because it is well-known allowing our results to be compared with other published work. These results also demonstrate the usefulness of query-time sampling (the operators and query optimizer transformation rules that we implemented). Figure 13a and Figure 13b show the change in performance and answers for all 22 queries in TPC-H.

The results use an input size of 100GB, i.e., a scale factor of 100; the data was generated with a custom data generator [14] which differs from the standard datagen in one way: instead of generating entries distributed uniformly at random, we skewed the frequency distribution with a zipf factor of 2. We executed the queries on a shared production data-parallel cluster. Figure 13a presents the median, minimum and average values over many different executions. Our metrics are: *latency* which is the job completion time and *total compute hours* which is the sum over all tasks in the job the execution time of the tasks.

Figure 13a shows that roughly 8 out of the 22 queries in TPC-H, specifically $\{2, 3, 10, 11, 13, 16, 18, 21\}$ are unsampled; as the figure shows, there is no appreciable change in the performance metrics for these queries; the changes visible represent the performance variability in our cluster due to contention from other jobs running in the cluster. These jobs do not benefit from sampling for a few different reasons, the most common reason is that they require stratification on such a large column set that samplers have to pass the entire input.

All but one of the other 14 queries receive sampled query expressions and improve; the exception is query #6 which receives a sampled plan with a 1% sampling rate but very little *work* remains after the sampler and so the cost of executing the sampler is not counter-weighted by improvements in later operations. Five of the queries (specifically $\{5, 7, 8, 9, 17\}$) improve substantially on both metrics: latency and total compute hours. Most of the other sampled queries also improve substantially in the total compute hours that they use, i.e., their processing cost substantially decreases but the query latency only improves moderately. This is primarily because when sampling reduces the data in flight subsequent parallel joins choose broadcast join implementations instead of pair (hashed) joins [15]; however doing so adds to the job's critical path because an aggregation step is needed to reduce the degree of parallelism of the sampled input and longer critical paths increase job latency.

Figure 13b shows the change in answer quality; our metrics are the fraction of missed rows in the answer, denoted as *missed groups* and the average over multiple aggregates of the average relative error across the various groups denoted as *AvgAvgRelErr*, the average over multiple aggregates of the maximum relative error across groups denoted as *AvgMaxRelErr* and the average over multiple aggregates the ratio of the average absolute error to the average true value. An example can help explain these metrics. Assume that the actual answer is $\{\texttt{X}, 10, 10\}, \{\texttt{Y}, 20, 2\}, \{\texttt{Z}, 30, 3\}$ and that the sampled answer is $\{\texttt{X}, 10.5, 11\}, \{\texttt{Y}, 21, 1\}$, then the fraction of missed groups is $\frac{1}{3} = 0.33$, `AvgAvgRelErr` $= 0.19$, `AvgMaxRelErr` $= 0.28$, `Avg(AvgAbsOverAvgTrueV)` $= 0.11$. Observe that the relative error metrics magnify small errors when the true values are small ($2 \rightarrow 1$ is a 50% error) but the last metric which divides the average absolute error value of an aggregate across groups by the average true value of the aggregate across groups is more robust.

As Figure 13b shows, none of the queries miss groups. Moreover, all aggregate error metrics are small for most of the queries, i.e., about 5% for 17 of the 22 queries; for most of the other queries, we see that the only large error metric is `AvgMaxRelErr` (shown in blue) but even in those case the average absolute value over true value (shown in orange) is small indicating that the error is limited to groups with small aggregate values. Overall, we conclude that the change in TPC-H answer quality is insignificant.

4.2 Case studies from production

We next present a few case studies of production jobs that

Table 4: Results for the case study in §4.2.1.

Metric	Job Stats.		Change
	Orig.	Sampled	
Duration (hours)	10.2	6.9	−32.4%
Total Compute Hours	5117.8	3182.7	−37.8%
Bytes read from disk (TB)	339.8	341.6	+0.5%
Bytes written to disk (TB)	171.6	169.8	−1.0%
# Tasks	85540	85457	−0.1%

Table 5: Comparing the sampled output with the unsampled output for the case study in §4.2.1

# Groups	650583
`MissedGroupFract.`	0.00
# Aggregates	24
`AvgAvgAbsOverAvgTrueV`	1.30%
`AvgAvgRelErr`	17.7%

use samplers; all of the presented jobs recur at least once per day throughout the examined period.

4.2.1 Case Study 1: 'Aggregates over Sampled Relations'

Our first case study illustrates the use-case where samplers were used to reduce the size of input leading into a group-by and aggregation. This case also illustrates the typical complexity of such production queries.

Figure 14 shows the sampled plan in the middle with data flowing from top to bottom; each circle denotes a collection of tasks with the size of the circle denoting the number of tasks (in log scale) and the color of the circle denoting the average execution time of the tasks in that collection; the legend is shown on the left. Lines encode data dependence with the width of the line denoting the volume of data exchanged and the color (green or brown) denoting whether data has to be shuffled or not.

The job joins five inputs and computes various grouped aggregates. Two of the joins execute in the stage marked J_1 and two remaining joins are in the stage marked J_2. Before these joins, the inputs are extracted, partitioned and aggregated. J_2 also executes a local group-by and aggregate after all four joins complete. A few other aggregates are computed on smaller subsets of the inputs; these are not sampled. This job uses a distinct sampler inside J_2 before both joins; the stratification is over the columns in the group.

The distinct sampler was pushed to one of the inputs below both of the outer joins in J_2 because preceding operations on the other join inputs allowed the QO to infer that these joins are primary key – foreign key joins (e.g., `SELECT DISTINCT` c or `SELECT c, Agg() Group By c` where `c` is the join column) [5]. The sampler could not be pushed below the other two joins, i.e., into J_1 or earlier, because no such primary key relationship could be inferred.

The distinct sampler in J_2 uses a frequency cutoff of 100 and a sampling probability of 0.1, i.e., $f = 100, p = 0.1$. In an example job, the sampler takes as input $9.7 * 10^{11}$ rows, outputs $2.0 * 10^{11}$ rows and runs in parallel in 2500 tasks.

Accuracy numbers, shown in Table 5, were considered adequate by the application developer.

Using the sampler does not result in a substantially different query plan, in this case, because the output of J_2 is roughly as large as in the unsampled plan, shown on the right, because in both cases a local group-by and aggregate executes in J_2 after the joins.

[5]Specifically, Rule-D3 from Table 3 was used here.

Table 6: Results for the case study in §4.2.2.

Metric	Job Stats.		Change
	Orig.	Sampled	
Duration (hours)	0.396	0.231	−41.7%
Total Compute Hours	67.28	42.65	−36.6%
Bytes read from disk (TB)	16.23	12.05	−25.8%
Bytes written to disk (TB)	11.30	6.90	−38.9%
# Tasks	9961	9961	0%

Table 7: Results for the case study in §4.2.3.

Metric	Job Stats.		Change
	Orig.	Sampled	
Duration (hours)	6.29	4.97	−21.0%
Total Compute Hours	21903.5	16467.4	−24.8%
Bytes read from disk (TB)	1037	1211	+16.8%
Bytes written to disk (TB)	446	764	+71.3%
# Tasks	668064	419064	−37.3%

However, as shown in Table 4, the sampled plan is substantially faster because the two large relations being outerjoined in J_2 after the sampler now execute on many fewer rows. Table 4 shows that the sampled version of the job consumes almost 2000 fewer cluster hours and finishes roughly 3 hours earlier. A typical single core VM on Azure today costs about 3 cents/ hour [2] and so even a conservative estimate that does not charge for storage or for the value-add from data-parallel services results in an annual savings of about 22,000$ by using samplers in this job [6]. Of course, the faster job completion time helps as well and we believe that the actual cost savings can be up to an order of magnitude larger when actual costs are considered. Such case studies lead us to strongly believe that the batched log analytics jobs that are prevalent in production big-data clusters can benefit substantially from query-time sampling.

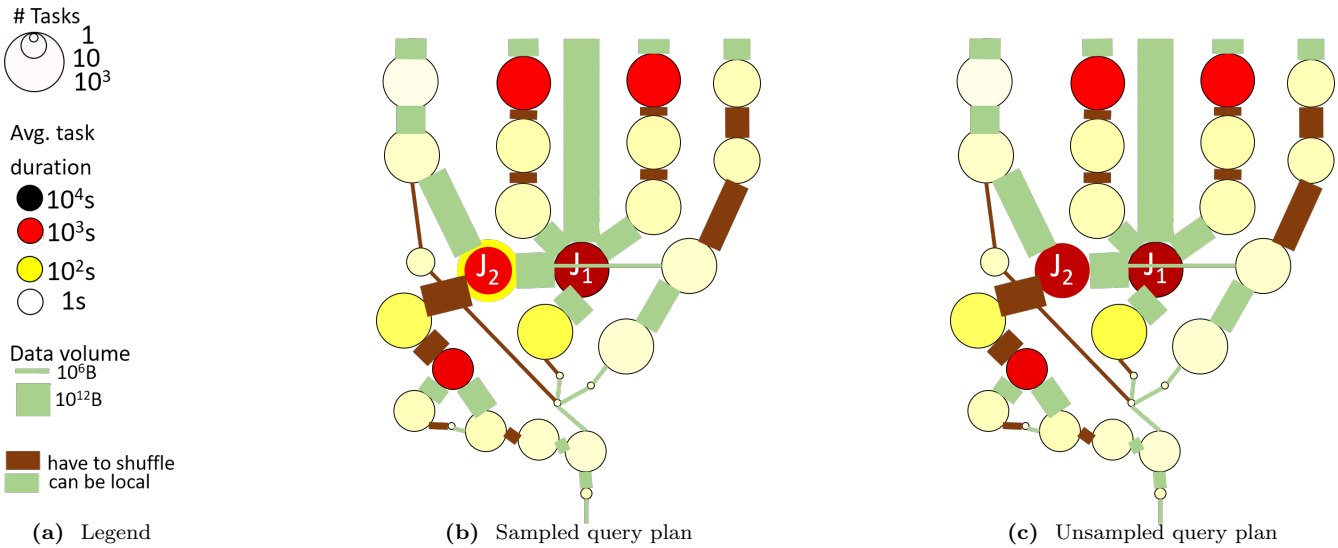
4.2.2 Case Study 2: 'Explicit Sampling'

Figure 15 and Table 6 show an instance of explicit sampling; the plan on the left uses a uniform sample with probability $p = 2 * 10^{-6}$. The sampler reads $7.0 * 10^9$ rows and outputs $1.5 * 10^4$ rows. Just for comparison, we show the unsampled plan on the right. As Table 6 shows the sampled plan is substantially faster because many tasks work on the much smaller sampled relation. Note however a clear difference from the previous case study (§4.2.1) where the sampled plan output is comparable to the output of the unsampled plan; here, the user query executes only on a sample (for reasons that are not apparent from examining the query) and so the output is not comparable to the output of the unsampled plan.

4.2.3 Case Study 3: Explicit partition before sample

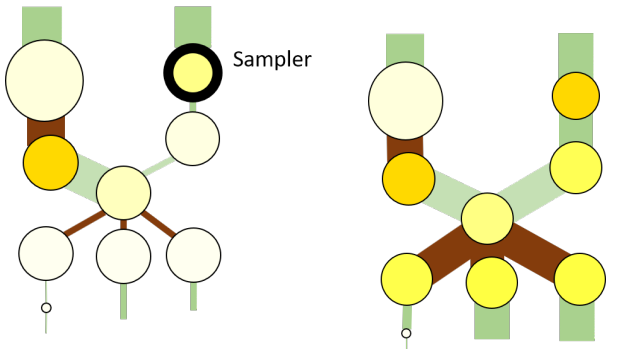
We present a second case where a distinct sampler was used to reduce the cost and latency of a query that computes grouped aggregates to highlight an interesting aspect of the distinct sampler. In this case, the distinct sampler was used with a frequency cut-off of $f = 100$, a $p = 0.3$ sampling probability and a group that consisted of over 20 different attributes, i.e., $|\mathcal{D}| > 20$. The sampler ran in parallel in 15000 tasks over an input of $4.5 * 10^{10}$ rows and produced an output of $4.5 * 10^{10}$ rows; i.e., the sampler passed all rows! The group (column set $\mathcal{D}$) was not a primary key but the distinct sampler was unable to reduce data because of the excessive parallelism; none of the groups had more than $f *$

[6]365 days/ year * 2000 cluster hrs/day * 0.03$/hr = 21900$/year



(a) Legend **(b)** Sampled query plan **(c)** Unsampled query plan

Figure 14: Query plans with and without the sampler for the case study in §4.2.1, 'Aggregates over Sampled Relations'.



(a) Sampled query **(b)** Unsampled counterpart

Figure 15: Query plans with and without the sampler for the case study in §4.2.2.

15000 rows and because the rows were arbitrarily distributed among the tasks, none of the tasks observed over $f = 100$ rows for any group. We advised the application developer to partition the input to the sampler on the group columnset. With this change, each sampler task sees disjoint subsets of groups allowing the sampler to output only $2.9 * 10^{10}$ rows; a data reduction of $\sim 34\%$. In spite of the additional cost to partition sampler input, Table 7 shows that both job latency and the total compute hours improve because substantial work occurs after the sampler [7].

5. DISCUSSION

Experiences engaging with users: In addition to the experiences noted in §1, we briefly mention experiences when users resisted using sampled plans:

- Users were more resistant to use sampling when a query's output was consumed by other groups because any changes in answer quality would now have to be cleared among many different groups.
- Whether or not users will use approximations, in some sense, appears akin to a religious belief; users who hesitate initially often remain unconvinced with logical explanations.
- Efficiency mandates which force groups to reduce their cluster usages (and long queues in backlogged clusters) were often a good motivation for users to consider approximations. However, especially for legacy scripts, we found that individual script owners lack the time and motivation to change their scripts. Tools that recommend how to change scripts (e.g., by picking appropriate sampler parameters) can help here.

Next steps: As a next step, we are focusing on automatically injecting sampler operators into recurring job queries where fine-grained data statistics, e.g., at the granularity of sub-expressions, can be obtained from previous job executions [15]. Other directions include: exploring richer set of sampler pushdown rules, considering cascades of samplers and extending the language's compiler to automatically rewrite aggregates add confidence intervals or other aposteriori error estimates.

6. RELATED WORK

Much research has focused on approximating queries; we defer a broader discussion to prior works [16, 22, 40, 31]. Almost every commercial database supports uniform sampling. Here, we concentrate on practical usages of approximations.

Using sketches to support specific aggregates has recently become common place. In particular, different data platforms including BigQuery [3], Oracle [7] and SQL Server [1] now support an approximate form of `COUNT DISTINCT` by using the HyperLogLog (HLL) sketch [25]. The HLL sketch is compact and can be constructed in parallel and merged leading to substantial speed-up. However, even simple extensions cannot be easily supported. For example, the HLL sketch cannot support predicates [8]. In contrast, the distinct

[7]Note the increase in the bytes to/from disk due to the extra partitioning.

[8]A HLL sketch on attribute c cannot be used to compute `SELECT (COUNT DISTINCT c) WHERE pred = 0`

sampler (e.g., with $f = 1,\ p = 0$) presents a more general alternative. Because the distinct sampler can be pushed below selections (e.g., by expanding the set of columns to stratify on), it can be more performant than HLL.

Sketches for percentiles, t-digest [11] in particular, are also used in production (e.g., at Netflix) although we are unaware of data platforms that support t-digests natively.

SnappyData [10, 38] uses differently stratified input samples in a way that is similar to [16]. We believe that input samples are very useful in some cases, e.g., when the workload has simple queries (e.g., only foreign key joins, no user-defined operations) which are known apriori and immutable datasets. As discussed earlier, input samples are not suitable widely. A key reason is that whereas query-time sampling can execute samplers any where in the query plan and in particular after selective predicates or other complex operations, input samples are constrained to only sample the input. VerdictDB [40] also only supports input samples; its key insight is that the samples can be stored in any underlying DBMS system and that queries can be rewritten to use these samples in a middleware layer.

7. CONCLUDING REMARKS

We have presented a first detailed longitudinal study of over tens of thousands of production queries that use approximations in Microsoft's big-data clusters. Our findings suggest new use-cases for sampler operators; we find that sampler pushdown rules improve plan performance while respecting the user-specified accuracy requirements. Our results show that the latency and cost vs. accuracy tradeoff from using approximations for production queries in Microsoft's big-data clusters is appealing, specifically for batch queries that recur. However, ceding control on accuracy to the query writer raises the bar for adoption because users have to identify the samplers and parameters that are appropriate for their query. The holy grail of automatically obtaining adequate answers for complex queries on large datasets at interactive timescales remains open.

8. REFERENCES

[1] Approx_count_distinct (transact-sql). `https://bit.ly/2Gyxgwa`.

[2] Azure: Virtual machines pricing. `https://bit.ly/2iKC7iq`.

[3] Bigquery: Hyperloglog++ functions in standard sql. `https://bit.ly/2V1gDfA`.

[4] C++ mersenne twister engine. `https://bit.ly/2EcqBp0`.

[5] Integer hash function. `https://bit.ly/1lcfJIt`.

[6] Logical processing order of the select statement. `https://bit.ly/2KMqRwJ`.

[7] Quick distinct count in oracle database 12cr1 (12.1.0.2). `https://bit.ly/2EldEJD`.

[8] Sample clause in ADLA. `https://bit.ly/2GJ30xS`.

[9] Sample expression in ADLA. `https://bit.ly/2S1qe4N`.

[10] Snappydata. `https://www.snappydata.io/`.

[11] T-digest. `https://github.com/tdunning/t-digest`.

[12] TPC-H Benchmark. `http://www.tpc.org/tpch`.

[13] TPC-DS Benchmark. `http://bit.ly/1J6uDap`, 2012.

[14] Program for tpc-h data generation with skew. `https://bit.ly/2wvdNVo`, 2016.

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

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

[17] A. Appleby. Murmurhash3. `https://bit.ly/2PsDyPy`.

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

[19] P. G. Brown and P. J. Haas. Techniques for warehousing of sample data. In *ICDE*, 2006.

[20] R. Chaiken et al. Scope: Easy and efficient parallel processing of massive data sets. *PVLDB*, 1(2):1265–1276, 2008.

[21] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *SIGMOD*, 2017.

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

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

[24] B. Ding et al. Sample + seek: Approximating aggregates with distribution precision guarantee. In *SIGMOD*, 2016.

[25] P. Flajolet et al. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. In *Analysis of Algorithms*, 2007.

[26] R. Gemulla et al. A dip in the reservoir: Maintaining sample synopses of evolving datasets. In *VLDB*, 2006.

[27] R. Gemulla and W. Lehner. Deferred maintenance of disk-based random samples. In *EDBT*, 2006.

[28] P. B. Gibbons and Y. Matias. New sampling-based summary statistics for improving approximate query answers. In *SIGMOD*, 1998.

[29] A. Greenberg et al. VL2: A Scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[30] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, 1999.

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

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

[33] S. Kandula. Errata and proofs for "Quickr". Technical Report MSR-TR-2017-14, 2017.

[34] S. Kandula et al. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, 2016.

[35] F. Li, B. Wu, K. Yi, and Z. Zhao. Wander join: Online aggregation via random walks. In *SIGMOD*, 2016.

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

[37] S.-J. Min, C. Iancu, and K. Yelick. Hierarchical work stealing on many-core clusters. In *PGAS*, 2011.

[38] B. Mozafari et al. Snappydata: A unified cluster for streaming, transactions, and interactive analytics. In *CIDR*, 2017.

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

[40] Y. Park et al. Verdictdb: Universalizing approximate query processing. In *SIGMOD*, 2018.