Approximate Partition Selection for Big-Data Workloads using Summary Statistics

Kexin Rong†*, Yao Lu†, Peter Bailis*, Srikanth Kandula†, Philip Levis*
Microsoft†, Stanford*

ABSTRACT

Many big-data clusters store data in large partitions that support access at a coarse, partition-level granularity. As a result, approximate query processing via row-level sampling is inefficient, often requiring reads of many partitions. In this work, we seek to answer queries quickly and approximately by reading a subset of the data partitions and combining partial answers in a weighted manner without modifying the data layout. We illustrate how to efficiently perform this query processing using a set of pre-computed summary statistics, which inform the choice of partition subset and weights. We develop novel means of using the statistics to assess the similarity and importance of partitions. Our experiments on several datasets and data layouts demonstrate that to achieve the same relative error compared to uniform partition sampling, our techniques offer from 2.7× to 70× reduction in the number of partitions read, and the statistics stored per partition require fewer than 100KB.

1. INTRODUCTION

Approximate Query Processing (AQP) systems allow users to trade off between accuracy and query execution speed. In applications such as data exploration and visualization, this trade-off is not only acceptable but is often desirable. Sampling is a common approximation technique, wherein the query is evaluated on a subset of the data, and much of the literature focuses on row-level samples [13, 16, 23].

When data is stored in media that does not support random access (e.g., flat files in data lakes and columnar stores [1, 53]), constructing a row-level sample can be as expensive as scanning the entire dataset. For example, if data is split into partitions with 100 rows, a 1% uniform row sample would in expectation require fetching 64% ($1-0.99^{100}$) of the partitions; a 10% uniform row sample would touch almost all partitions. As a result, recent work from a production AQP system shows that row-level sampling only offers significant speedups for complex queries where substantial query processing remains after the sampling [42].

In contrast to row-level sampling, the I/O cost of constructing a *partition-level* sample is proportional to the sampling fraction[1]. In our example above, a 1% partition-level sample would only read 1% of the data. We are especially interested in big data clusters, where data is stored in chunks of tens to hundreds of megabytes, instead of disk blocks or pages which are typically a few kilobytes [34, 53]. Partition-level sampling is already used in production due to its appealing performance: commercial databases create statistics using partition samples [5, 30] and several Big Data stores allow sampling partitions of tables [2, 6, 8].

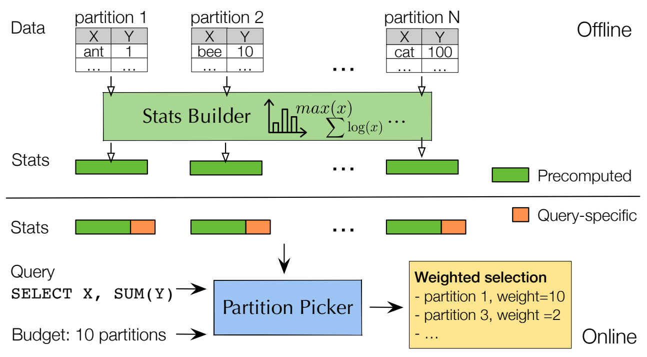


Figure 1: Our system PS^3 makes novel use of summary statistics to perform importance and similarity-aware sampling of partitions.

However, since all or none of the rows in a partition are included in the sample, the correlation between rows (e.g., due to layout) can lead to inaccurate answers. A key challenge remains in how to construct partition-level samples that can answer a given query accurately. A uniformly random partition-level sample does not make a representative sample of the dataset unless the rows are randomly distributed among partitions [24], which happens rarely in practice [20]. In addition, even a uniform random sample of rows can miss rare groups in the answer or miss the rows that contribute substantially to `SUM`-like aggregates. It is not known how to compute stratified [13] or measure-biased [32] samples over partitions, which helps with queries that have group-by's and complex aggregates.

In this work, we introduce PS^3 (Partition Selection with Summary Statistics), a system that supports AQP via weighted partition selection (Figure 1). Our primary use case is in large-scale production query processing systems

[1]In the paper, we use "partition" to refer to the finest granularity at which the storage layer maintains statistics.

such as Spark [15], F1 [52], SCOPE [21] where queries *only read* and datasets are *bulk appended*. Our goal is to minimize the approximation error given a sampling budget, or fraction of data that can be read. Motivated by observations from production clusters at Microsoft and in the literature that many datasets remain in the order that they were ingested [43], PS^3 does not require any specific layout or re-partitioning of data. Instead of storing precomputed samples [14, 16, 22], which requires significant storage budgets to offer good approximations for a wide range of queries [25, 44], PS^3 performs sampling exclusively during query optimization. Finally, similar to the query scope studied in prior work [14, 48, 54], PS^3 supports single-table queries with `SUM, COUNT(*), AVG` aggregates, `GROUP BY` on columnsets with moderate distinctiveness, predicates that are conjunctions, disjunctions or negations over single-column clauses.

To select partitions that are most relevant to a query, PS^3 leverages the insight that partition level summary statistics are relatively inexpensive to compute and store. The key question is which statistics to use. Systems such as Spark SQL and ZoneMaps already maintain statistics such as maximum and minimum values of a column to assist in query optimization [43]. Following similar design considerations, we look for statistics with small space requirements that can be computed for each partition in one pass at ingest time. For functionality, we look for statistics that are discriminative enough to support decisions such as whether the partition contributes disproportionally large values of the aggregates. We propose such a set of statistics for partition sampling – measures, heavy hitters, distinct values, and histograms – which include but expand on conventional catalog-level statistics. The total storage overhead scales with the number of partitions instead of with the dataset size and we only maintain single-column statistics to keep the overhead low. The resulting storage overhead can be orders of magnitudes smaller than approaches using auxiliary indices to reduce the cost of random access [32]. While this set of statistics is by no means complete, we show that each type of statistics contributes to the sampling performance, and, in aggregate, delivers effective AQP results.

We illustrate three ways of leveraging summary statistics to help with partition selection. First, if we knew which partitions contribute more to the query answer, we could sample these partitions more frequently. While it is challenging to manually design rules that relate summary statistics to partition contribution, a model can *learn* how much summary statistics matter through examples. Inspired by prior work which uses learning techniques to improve the sampling efficiency for counting queries on row-level samples [56], we propose a learned importance-style sampling algorithm that works on aggregate queries with `GROUP BY` clauses and on partitions. The summary statistics serve as a natural feature representation for partitions, from which we can train models offline to learn a mapping from summary statistics to the relative importance of a partition. During query optimization, we use the trained models to classify partitions into several groups of importance, and split the sampling budget across groups such that the more important groups get a greater proportion of the budget. The training overhead is a one time cost for each dataset and workload, and, for high-value datasets in clusters that are frequently queried, this overhead is amortized over time.

In addition, we leverage the redundancy and skewness of the partitions for further optimization. For two partitions that output similar answers to an input query, it suffices to only include one of them in the sample. While directly comparing the contents of the two partitions is expensive, we can use the query-specific summary statistics as a proxy for the similarity between partitions. We also observe that datasets commonly exhibit significant skew in practice. For example, in a prototypical production service request log dataset at Microsoft, the most popular application version out of the 167 distinct versions accounts for almost half of the dataset. Inspired by prior works in AQP that recognize the importance of outliers [16, 23], we use summary statistics such as the occurrences of heavy hitters in a partition to identify a small number of partitions that are likely to contain rare groups and dedicate a portion of the sampling budget to evaluate these partitions exactly.

In summary, this paper makes the following contributions:

1. We introduce PS^3, a system that makes novel uses of summary statistics to perform weighted partition selection for many popular queries. Given the query semantics, summary statistics and a sampling budget, the system intelligently combines a few sampling techniques to produce a set of partitions to sample and the weight of each partition.

2. We propose a set of lightweight sketches for data partitions that are not only practical to implement, but can also produce rich partition summary statistics. While the sketches are well known, this is the first time the statistics are used for weighted partition selection.

3. We evaluate on a number of real-world datasets with real and synthetic workloads. Our evaluation shows that each component of PS^3 contributes meaningfully to the final accuracy and together, the system outperforms alternatives across datasets and layouts, delivering from 2.7× to 70× reduction in data read given the same error compared to uniform partition sampling.

2. SYSTEM OVERVIEW

In this section, we give an overview of PS^3, including its major design considerations, supported queries, inputs, and outputs, and the problem statement.

2.1 Design Considerations

We highlight a few design considerations in the system.

Layout Agnostic. A random data layout would make the partition selection problem trivial, but maintaining a random layout requires additional efforts and rarely happens in practice [20]. In read-only or append-only data stores, it is also expensive to modify the data layout. As a result, we observe that in practice, many datasets simply remain in the order that they were ingested in the cluster. In addition, prior work [54] has shown that it is challenging and sometimes impossible to find a partitioning scheme that enables good data skipping for arbitrary input queries. Therefore, instead of requiring re-partitioning or random layout, PS^3 explicitly chooses to keep data in situ and tries to make the best out of the given data layout. We show that PS^3 can work across different data layouts in the evaluation (§ 5.5.1).

Sampling on a single table. To perform joins effectively, prior work [44] has shown that sampling each input relation independently is not enough and that the joint distribution must be taken into account. Handling the correlations between join tables at the partition level is another research problem on its own, and is outside the scope of this paper. However, sampling on a single table can still offer non-trivial performance improvements even for queries that involve joining multiple tables. For example, in key–foreign key joins, fact tables are often much larger compared to dimension tables. Sampling the fact table, therefore, already gets us most of the gains.

Generalization. Prior works make various trade-offs between the efficiency and the generality of the queries that they support, ranging from having access to the entire workload [54] to being workload agnostic [38]. Our system falls in the middle of the spectrum, where we make assumptions about the structure and distribution of the query workload. Specifically, we assume that the set of columns used in `GROUP BY`s and the aggregate functions are known apriori, with the scope defined in § 2.2; predicates can take any form that fits under the defined scope and we do not assume we have access to the exact set of predicates used. We assume that a workload consists of queries made of an arbitrary combination of aggregates, group bys and predicates from the scope of interest. PS^3 is trained per data layout and workload, and generalizes to unseen queries sampled from the same distribution as the training workload. Overall, our system is best suited for commonly occurring queries and should be retrained in case of major changes in query workloads such as the introduction of unseen group by columns.

We do not consider generalization to unseen data layouts or datasets and we view broader generalization as an exciting area for future work (§ 7). Since most summary statistics are computed per column, different datasets might not share any common statistics. Even for the same dataset, the importance of summary statistics can vary across data layouts. For example, the mean of column X can distinguish partitions in a layout where the dataset is sorted by X, but may provide no information in a random layout.

2.2 Supported Queries

In this section, we define the scope of queries that PS^3 supports. We support queries with an arbitrary combination of aggregates, predicates and group bys. Although we do not directly support nested queries, many queries can be flattened using intermediate views [36]. Our techniques can also be used directly on the inner queries. Overall, our query scope covers 11 out of 22 queries in the TPC-H workload (Appendix A.1).

- `Aggregates`. We support `SUM` and `COUNT(*)` (hence `AVG`) aggregates on columns as well as simple linear projections of columns in the select clause. The projections include simple arithmetic operations (`+, -`) on one or more columns in the table[2]. We also support a subset of aggregates with `CASE` conditions that can be rewritten as an aggregate over a predicate.
- `Predicates`. Predicates include conjunctions, disjunctions and negations over the clauses of the form c op v, where c denotes a column, op an operation and v a value. We support equality and inequality comparisons on numerical and date columns, equality check with a value as well as the `IN` operator for string and categorical columns as clauses.
- `Groups`. We support `GROUP BY` clauses on one or more stored attributes[3]. We do not support `GROUP BY` on columns with large cardinality since there is little gain from answering highly distinct queries over samples; one could either hardly perform any sampling without missing groups, or would only care about a limited number of groups with large aggregate values (e.g., `TOP` queries), which is out of the scope of this paper.
- `Joins`. Queries containing key–foreign joins can be supported as queries over the corresponding denormalized table. For simplicity, our discussion in this paper is based on a denormalized table.

In the TPC-H workload, 16 out of the 22 queries can be rewritten on a denormalized table and 11 out of the 16 are supported by our query scope. For the 5 that are not supported, 4 involve group bys on high cardinality columns and 1 involves the `MAX` aggregate. A number of prior work have also studied similar query scopes [14, 48, 54].

2.3 Inputs and Outputs

PS^3 consists of two main components: the statistics builder and the partition picker (Figure 1). In this section, we give an overview of the inputs and outputs of each component during preprocessing and query time.

2.3.1 Statistics Builder

Preparation. The statistics builder takes a partition as input and outputs a number of lightweight sketches for each partition. The sketches are stored separately from the partitions. We describe the sketches used in detail, including the time and space complexity for constructing and storing the sketches in § 3.1.

Query Time. During query optimization, one can access the sketches *without* touching the raw data. Given an input query, the statistics builder combines pre-computed column statistics with query-specific statistics computed using the stored sketches and produces a set of summary statistics for each partition and for each column used in the query.

2.3.2 Partition Picker

Preparation. In the preparation phase, the picker takes a specification of workload in the form of a list of aggregate functions and columnsets that are used in the `GROUP BY`. We can sample a query from the workload by combining randomly generated predicates and randomly selected aggregate functions and group by columnsets (0 or 1) from the specification. For each sampled query, we compute the summary statistics as well as the answer to the query on each partition as the training data, which the picker uses to learn the relevance of different summary statistics. The training is a one time cost and we train one model for each workload to be used for all test queries. We elaborate on the design of the picker in Section 4.

Query Time. The picker takes an input query, summary statistics and a sampling budget as inputs, and outputs a list of partitions to sample, as well as the weight of each partition in the sample. This has a net effect of replacing

[2]We also support the multiply and divide operations in some cases using statistics computed over the logs of the columns.

[3]To support derived attributes, we make a new column from the derived attribute and store its summary statistics

Table 1: Per partition, the time and space overheads to construct and store sketches for a dataset with R_b rows in each partition.

Sketch	Construction	Storage
Histograms	$O(R_b \log R_b)$	$O(\#buckets)$
Measures	$O(R_b)$	$O(1)$
AKMV	$O(R_b)$	$O(k)$
Heavy Hitter	$O(R_b)$	$O(\frac{1}{support})$

a table in the query execution plan with a set of weighted partition choices with a small overhead (Table 5). Prior work [44] has suggested ways to augment query execution to handle weights for each row.

2.4 Problem Statement

Let N be the total number of partitions and M be the dimension of the summary statistics. For an aggregation query Q, let G be the set of groups in the answer to Q. For each group $g \in G$, denote the aggregate values for the group as $\mathbf{A_g} \in \mathbb{R}^d$, where d is the number of the aggregates. Denote the aggregates for group g on partition i as $\mathbf{A_{g,i}}$.

Given the input query Q, the summary statistics $F \in \mathbb{R}^{N \times M}$ as well as sampling budget n in the form of number of partitions to read, our system returns a set of weighted partition choices $S = \{(p_1, w_1), (p_2, w_2), ..., (p_n, w_n)\}$. The approximate answer $\tilde{\mathbf{A}}_\mathbf{g}$ of group g for Q is computed by $\tilde{\mathbf{A}}_\mathbf{g} = \sum_{j=1}^{n} w_j \mathbf{A_{g,p_j}}, \forall g \in G$.

Our goal is to produce the set of weighted partition choice S such that $\tilde{\mathbf{A}}_\mathbf{g}$ is a good approximation of the true answer $\mathbf{A_g}$ for all groups $g \in G$. To assess the approximation quality across groups and aggregates that are of different sizes and magnitudes, we measure absolute and relative error, as well as the percentage of groups that are missed in the estimate.

3. PARTITION SUMMARY STATISTICS

The high-level insight of our approach is that we want to differentiate partitions based on their contribution to the query answer, and that the contribution is estimated using a rich set of summary statistics on the partitions. As a simple example, for `SUM`-type aggregates, partitions with a higher average value of the aggregate should be preferred, all else being equal. We are unaware of prior work that uses partition level summary statistics for performing non-uniform partition-level sampling. In this section, we describe the design and implementation of the summary statistics.

3.1 Lightweight Sketches

Our primary use case, similar to columnar databases, is read-only or append-only stores. Summary statistics are constructed for each new data partition when the partition is sealed. The necessary data statistics should be simple, small in size and can be computed incrementally in one pass over data. The necessary statistics should also be discriminative enough to set partitions apart and rich enough to support sampling decisions such as estimating the number of rows that pass the predicate in a partition. We opt to use only single-column statistics to keep the memory overhead light, although more expensive statistics such as multi-column histograms can help estimate selectivity more accurately. The design considerations lead us to the following sketches:

Table 2: Summary statistics and the sketches used to compute them. Selectivity is computed per query and all other statistics is computed per column.

Summary Statistics	Sketch
$\overline{x}$, $min(x)$, $max(x)$, $\overline{x^2}$, $std(x)$	Measures
$\overline{\log(x)}$, $\overline{\log(x)^2}$, $min(\log(x))$, $max(\log(x))$	Measures
number of distinct values	AKMV
avg/max/min/sum freq. of distinct values	AKMV
# hh, avg/max freq. of hh	Heavy Hitter
occurrence bitmap of heavy hitters	Heavy Hitter
selectivity	Histogram

- **Measures:** Minimum, maximum, as well as first and second moments are stored for each numeric column. For columns whose value is always positive, we also store measures on the log transformed column.
- **Histogram:** We construct equal-depth histograms for each column. For string columns, the histogram is built over hashes of the strings. By default, each histogram has 10 buckets.
- **AKMV:** We use an AKMV (K-Minimum Values) sketch to estimate the number of distinct values [19]. The sketch keeps track of the k minimum hashed values of a column and the number of times these values appeared in the partition. We use $k = 128$ by default.
- **Heavy Hitter:** We maintain a dictionary of heavy hitters and their frequencies for each column in the partition using lossy counting [46]. By default, we only track heavy hitters that appear in at least 1% of the rows, so the dictionary has at most 100 items.

Table 1 summarizes the time complexity to construct the sketches and the space overhead to store them, ignoring small logarithmic factors. The sketches can be constructed in parallel for each partition. We do not claim that the above choices make a complete set of sketches that should be used for the purpose of partition selection. Our point is that these are a set of inexpensive sketches that can be easily deployed or might have already been maintained in big-data systems [43], and that they can be used in new ways to improve partition sampling.

3.2 Summary Statistics as Features

Given the set of sketches, we compute summary statistics for each partition, which can be used as *feature vectors* to discriminate partitions based on their contribution to the answer of a given query. The features consist of two parts: pre-computed per column features and query-specific selectivity estimates (Table 2). We apply a query-dependent mask on the pre-computed column features: features associated with columns that are not used in the query are set to zero. In addition, for categorical columns where the measure based sketches do not apply, we set the corresponding features to zero. The schema of the feature vector is determined entirely by the schema of the table, so queries on the same table share the feature vector schema.

Overall, there are four types of features based on the underlying sketches that generate them: measures, heavy hitters, distinct values and selectivity. Each type of feature captures different information about the partitions and the queries. Measures help identify partitions with disproportionally large values of the aggregates; heavy hitters and distinct values help discriminate partitions from each other and selectivity helps assess the impact of the predicates. We found that all types of features are useful in PS^3 but the relative importance of each varies across datasets (§ 5.4.2).

Extracting features from sketches is, in general, straightforward; here, we discuss two interesting cases.

Occurrence Bitmap. We found that it is not only helpful to know the number of heavy hitters, but also *which* heavy hitters are present in the partition. To do so, we collect a set of k global heavy hitters for a column by combining the heavy hitters from each partition. For each partition, we compute a bitmap of size k, each bit representing whether the corresponding global heavy hitter is also a heavy hitter in the current partition. The feature is only computed for grouping columns and we cap k at 25 for each column.

Selectivity Estimates. The selectivity estimate is a real number between 0 and 1, designed to reflect the fraction of rows in the partition that satisfies the query predicate. The estimate supports predicates defined in our query scope (§ 2.2) and is derived using histograms over individual columns. We describe a few special cases below. If a string column has a small number of distinct values, each distinct value and its frequency is stored exactly; this can support regex-style textual filters on the string column (e.g. `'%promo%'`). We jointly evaluate predicates that use the same column (e.g., $X < 1$ or $X > 10$). We use the following four features to represent the selectivity of predicates which can be a conjunction or disjunction of individual clauses:

1. `selectivity_upper`: For `AND`s, the selectivity is at most the min of the selectivity of individual clauses; for `OR`s, the selectivity is at most 1 and at most the sum of selectivity of individual clauses.
2. `selectivity_indep`: This feature computes the selectivity assuming independence between predicate clauses. For `AND`s, the feature is the product of the selectivity for each individual clause; for `OR`s, the feature is the min of the selectivity of individual clauses.
3. `selectivity_min`, `selectivity_max`: We store the min and max of the selectivity of individual clauses.

If the upper bound of the selectivity is zero, the partition contains no rows that pass the predicate; if the upper bound is nonzero however, the partition can have zero or more rows that pass the predicate. In other words, as a classifier for identifying partitions that satisfy the predicate, `selectivity_upper`> 0 has perfect recall and uncertain precision. For simple predicates such as $X > 1$, the precision is 100%; for complicated predicates involving conjunctions and disjunctions over many clauses and columns (e.g., TPC-H Q19), the precision can be as low as 10%.

4. PARTITION PICKING

In this section, we describe PS^3's partition picker component and how it makes novel use of the summary statistics discussed above to realize weighted partition selection.

4.1 Picker Overview

To start, we give an overview of how our partition picker works. Recall that the picker takes a query, the summary statistics and a sampling budget as inputs, and outputs a list of partitions to evaluate the query on and the weight of each partition. Partial answers from the selected partitions are combined in a weighted manner, as described in § 2.4.

Algorithm 1 Partition Picker

```
Input: partition features $F$, sampling budget $n$, group-by
    columns gb_col, models regrs, decay rate $\alpha$
Output: part_choice: $[(p_1, w_1), (p_2, w_2), ..., (p_n, w_n)]$
1: outliers, inliers $\leftarrow$ OUTLIER($F$, gb_col)
2: $n_o \leftarrow$ outliers.size()
3: part_choice.add(outliers, $[1] * n_o$)
4: groups $\leftarrow$ IMPORTANCEGROUP($F$, inliers, regrs)
5: $n_c \leftarrow$ ALLOCATESAMPLES(groups, $n - n_o$, $\alpha$)
6: for $j \leftarrow 1, ...,$groups.size() do
7:     part_choice.add(CLUSTERING($F$[groups[i]], $n_c[i]$))
8: end for
```

Algorithm 1 describes the entire procedure. We first identify outlier partitions with rare groups using the procedure described in § 4.4. Each outlier partition has a weight of 1. We then use the trained models to classify the remaining partitions into groups of different importance, using the algorithm described in § 4.3. We allocate the remaining sampling budget across groups such that the sampling rate decreases by a factor of α from the i^{th} important to the $(i+1)^{th}$ important group. Finally, given a sample size and a set of partitions in each importance group, we select samples via clustering using the procedure described in § 4.2. An exemplar partition is selected from each cluster, and the weight of the exemplar equals the size of the cluster. We explain each component in detail in the following sections.

4.2 Sample via Clustering

We start by describing the sampling procedure (line 7 in Algorithm 1), which uses clustering to leverage the redundancy between partitions. We use feature vectors to compute a similarity score between partitions, which consequently enables us to choose dissimilar partitions as representatives of the dataset. In fact, identical partitions will have identical summary statistics, but the converse does not hold; having summary statistics on multiple columns as well as multiple statistics for each column makes it less likely that dissimilar partitions have identical summary statistics.

We propose to use *clustering* as a sampling strategy: given a sampling budget of n partitions, we perform clustering using feature vectors with a target number of n clusters; an exemplar partition is chosen per cluster, with an assigned weight equals the number of partitions in the cluster. Denote the answer to the query on cluster i's exemplar partition as A_i and the size of cluster i as s_i. The estimate of the query answer is given by $\tilde{A} = \sum_{i=1}^{n} s_i A_i$.

Concretely, we measure partition similarity using Euclidean distances of the feature vectors. We zero out features for unused columns in the query so they have no impact on the result; we also perform normalization such that the distance is not dominated by any single feature (Appendix B). Regarding the choice of the clustering algorithm, we experimented with KMeans and Agglomerative Clustering and found that they perform similarly. Finally, the cluster exemplar is selected by picking the partition whose feature vector has the smallest distance to the median feature vector of partitions in the clusters.

Our proposed scheme leads to a biased estimator that can be challenging to analyze. Specifically, given the median feature vector of a cluster, our estimator deterministically picks the partition that is closest to the median vector as the cluster exemplar. However, one could make a simple modification to unbias the estimator by selecting a random partition in the cluster as the exemplar instead. We have included an empirical comparison of the accuracy of the two estimators as well as a variance analysis for the unbiased estimator in Appendix D. We have empirically found that the proposed scheme outperforms its unbiased counterpart when the sampling budget is limited.

Clustering effectively leverages the redundancy between partitions, especially in cases when partitions have near identical features. Although there is no guard against an adversary, in practice, having a large and diverse set of summary statistics makes it naturally difficult for dissimilar partitions to be in the same cluster. Clusters play a similar role as strata in stratified sampling. The goal of clustering is to make partitions in the same stratum homogeneous such that the overall sampling variance is reduced. Finally, clustering results vary from query to query: the same partition can be in different clusters for different queries due to the changes in selectivity features and the query-dependent column masks.

Feature Selection. Clustering assumes that all features are equally relevant to partition similarity. To further improve the clustering performance, we perform feature selection via a "leave-one-out" style test. For example, consider a table with columns X, Y and features min, max. We compare the clustering performance on the training set using $\{min(X), max(X), min(Y), max(Y)\}$ as features to that from using only $\{max(X), max(Y)\}$ as features. If the latter gives a smaller error, we subsequently exclude the min feature for all columns from clustering. We greedily remove features until converging to a local optimal, at which point excluding any remaining features would hurt clustering performance. In an outer loop, we repeat the above greedy procedure multiple times, each time starting with a random ordering of the features. We provide the pseudo code of the procedure in Appendix B.1. Our experiments show that feature selection consistently improves clustering performance across datasets .

Limitations. We briefly discuss two failure cases for clustering in which PS^3 can fall back to random sampling, and include more details in Appendix B.1. First, clustering takes advantage of the redundancy among partitions. In the extreme case when the query groups by the primary key, no two partitions contribute similarly to the query and any downsampling would result in missed groups. As discussed in § 2.2, our focus is on queries where such redundancy exists. Second, queries with highly selective predicates might suffer from poor clustering performances. Since most features are computed on the entire partition, the features would no longer be representative of partition similarity if only a few rows satisfy the predicate in each partition.

4.3 Learned Importance-Style Sampling

While clustering helps select partitions that are dissimilar, it makes no distinction between partitions that contribute more to the query and partitions that contribute less. Ideally, we would want to sample the more important partitions more frequently to reduce the variance of the estimate [37].

The feature vectors can help assess partition contribution. Consider the query: `SELECT SUM(X), Y FROM table WHERE Z > 1 GROUP BY Y`. The subset of partitions that answer this query well should contain large values of `X`, many rows that satisfy the predicate and many distinct values of `Y`. Feature vectors are correlated with these desired properties: measure statistics (e.g. max, std) can help reveal large values of `X`, selectivity measures the fraction of the partition that is relevant to the query, and heavy hitter and distinct value statistics summarize the distribution of groups. However, it is challenging to manually quantify how much each feature matters for each query. In our example, it is unclear whether a partition with a high variance of `X` but few rows that match the predicate should be prioritized over a partition with low variance and many rows that match the predicate.

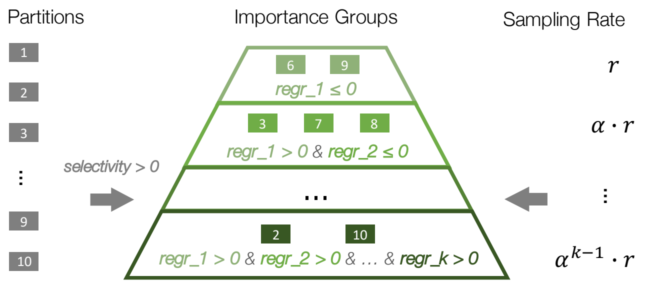


Figure 2: The trained regressors are used to classify input partitions into groups of different importance. The sampling rate decreases by a factor of $\alpha > 1$ from the i^{th} important to the $(i+1)^{th}$ important group.

While it is not obvious how to manually design rules that relate feature vectors to partition contribution, a model may *learn* to do so from examples. An intuitive design is to use partition features as inputs and predict partition weights as outputs, which turns out to be a non-traditional regression problem. The goal of the regressor is to assign a weight vector to N partitions such that the weighted partition choice produces a small approximation error. Given a sampling budget of n partitions, there are exponentially many choices of subsets of partitions of size n and the optimal choice is discontinuous on n[4]. In addition, the decision depends jointly on the *set* of partitions chosen; the weight assigned to one partition, for example, may depend on how many other partitions with nearly identical content are picked in the sample. Therefore, a simple, per partition regressor is unable to capture the combinatorial nature of the decision space. Existing solutions [18, 45] would require significantly more resources and we pursue a lightweight alternative instead.

Given the challenges to directly use learned models to predict sampling probabilities, we propose a design that utilizes the models indirectly for sample size allocation; similar observations were made for using learned models to improve row-level sampling designs for count queries [56]. We consider classifying partitions based on their *relative importance* to the query answer into a few importance groups, and apply multiplicatively increasing sampling probability to the more important groups. We detail each of these steps next.

Partition Contribution. We consider the "contribution" of a partition to the answer of a query as its largest relative contribution to any group and any aggregate in the answer.

[4]A small change in n can completely change the set of partitions to pick and the weights to assign to them.

Recall that we denote the aggregates for group $g \in G$ as $\mathbf{A_g} \in \mathbb{R}^d$, where d is the number of the aggregate functions, and the aggregates for group g on partition i are denoted as $\mathbf{A_{g,i}} \in \mathbb{R}^d$. Partition i's contribution is defined as: $\max_{g \in G} \max_{j=1}^{d}(\frac{\mathbf{A_{g,i}}[j]}{\mathbf{A_g}[j]})$. There are several alternative definitions of contribution, such as using the average instead of the max of the ratios, or using absolute values instead of the relatives. Among all variants, the max of the relatives is perhaps the most generous: it recognizes a partition's importance if it helps with *any* aggregates in *any* groups, and is not biased towards large groups or aggregates with large absolute values. We find that our simple definition above already leads to good empirically results.

Training. Given the partition contributions for all queries in the training data, we train a set of k models to distinguish the relative importance of partitions. When k is large, training the set of models is equivalent to solving the regression problem in which we are directly predicting partition contribution from the feature vector; when k is small, the training reduces to a simpler multiclass classification problem. The k models discretize partition contribution into $k+1$ bins, and we choose exponentially spaced bin boundaries: the number of partitions that satisfy the i^{th}model increase exponentially from the number of partitions that satisfy the $(i+1)^{th}$ model. In particular, the first model identifies all partitions that have non zero contribution to the query and the k^{th} model identifies partitions whose contribution is ranked in the top 1% of all partitions[5]. We use the `XGBoost` regressor as our base model, and provide additional details of the training in Appendix B.

Testing. During test time, we run partitions through a funnel that utilizes the set of trained models as filters and sort partitions into different groups of importance (Figure 2). The advantage of building a funnel is that it requires partitions to pass more filters as they advance to the more important groups, which help limit the impact of inaccurate models. We list the procedure in Algorithm 2. We start from all partitions with non zero `selectivity_upper` feature; as discussed in § 3.2, this filter has perfect recall but varying precision depending on the complexity of the predicates. We run the partitions through the first trained model, and move the ones that pass the model to the next stage in the funnel. We repeat this process, each time taking the partitions at the end of the funnel, running them through a more restrictive filter (regressor) and advance ones that pass the filter into the next stage until we run out of filters.

We then split the sampling budget such that more important groups get a greater proportion of the budget. We implement a sampling rate that decays by a factor of $\alpha > 1$ from the i^{th} important to the $(i+1)^{th}$ important group. We investigate the impact of the decay rate α in the sensitivity analysis (Appendix C.2). In general, increasing α improves the overall performance especially when the trained models are accurate, but the marginal benefit decreases as α becomes larger. If the trained models are completely random however, a larger α would increase the variance of the estimate. We have found that a decay rate of $\alpha = 2$ with $k = 4$ models works well across a range of datasets and layouts empirically. However, it is possible to fine-tune α for each dataset to further improve the performance and we leave the fine-tuning to future work.

[5]The small number of positive examples make it challenging to train an accurate model beyond 1%.

Algorithm 2 Group partitions by importance.

```
Input: partition features $F$
1: function ImportanceGroup($F$, parts, regressors)
2:     groups.add(FilterByPredicate($F$, parts))
3:     for regr $\in$ regressors do
4:         to_examine $\leftarrow$ groups[-1]
5:         to_pick $\leftarrow$ p $\in$ to_examine s.t. regr($F[p]$) $> 0$
6:         groups[-1] $\leftarrow$ to_examine.difference(to_pick)
7:         groups.add(to_pick)
8:     end for
9:     return groups
10: end function
```

4.4 Outliers

Finally, we observe that datasets often exhibit significant skew in practice (example in § 1). Prior work in AQP has shown that augmenting random samples with a small number of samples with outlying values or from rare groups helps reduce error caused by the skewness [16, 23]. We recognize the importance of handling outliers and allocate a small portion of the sampling budget for outlying partitions.

We are especially interested in partitions that contain a rare distribution of groups for `GROUP BY` queries. These partitions are not representative of other partitions and should therefore be excluded from clustering. To identify such partitions, we take advantage of the occurrence bitmap feature that tracks which heavy hitters are present in a partition. We put partitions with identical bitmap features for columns in the `GROUP BY` clause in the same group and consider a bitmap feature group outlying if its size is small both in absolute (< 10 partitions) and relative terms ($< 10\%$ the size of the largest group). For example, if there are 100 such bitmap feature groups and 10 partitions per group, we do not consider any group as outlying although the absolute size of each group is small. We allocate up to 10% of the sampling budget to evaluate outliers. We have empirically found that increasing the outlier budget further does not significantly improve with the performance for the outliers we consider. Exploring alternative ways to identify outliers could be an area for improvement for future works.

5. EVALUATION

In this section, we evaluate the empirical performance of PS^3. Experiments show that:

1. PS^3 consistently outperforms alternatives on a variety of real-world datasets, delivering $2.7-70\times$ reduction of data read to achieve the same average relative error compared to uniform partition sampling, with storage overhead ranging from 12KB to 103KB per partition.
2. Every component of PS^3 and every type of features contribute meaningfully to the final performance.
3. PS^3 works across datasets, partitioning schemes, partition counts and generalizes to unseen queries.

5.1 Methodology

In this subsection, we describe the experimental methodology, which includes the datasets, query generation, methods of comparison and error metrics.

5.1.1 Datasets

We evaluate on four real-world datasets that are summarized below. We include a specification of the table schema in Appendix A.

TPC-H*. Data is generated from a Zipfian distribution with skewness of 1 and a scale factor of 1000 [7]. We denormalize all tables against the *lineitem* table. The resulting table has 6B rows, with 14 numeric columns and 31 categorical columns. Data is sorted by column `L_SHIPDATE`.

TPC-DS*. *catalog_sales* table with a scale factor of 1 from TPC-DS, joined with dimensions tables *item*, *date_dim*, *promotion* and *customer_demographics*, with 4.3M rows, 21 numeric columns and 20 categorical columns. Data is sorted by columns `year`, `month` and `day`.

Aria. Production service request log at Microsoft with 10M rows, 7 numeric columns and 4 categorical columns [11, 33]. Data is sorted by categorical column `TenantId`.

KDD. KDD Cup'99 dataset on network intrusion detection with 4.8M rows, 27 numeric columns and 14 categorical columns [17]. Data is sorted by numeric column `count`.

By default we use a partition count of 1000, the smallest size from which partition elimination becomes interesting. The `TPC-H*` dataset (sf=1000) has 2844 partitions, with a partition size of about 2.5GB, consistent with the scale of the big-data workloads seen in practice. In the sensitivity analysis, we further investigate the effect of the partition count (§ 5.5.3), and data layouts on the performance (§ 5.5.1).

5.1.2 Query Set

To train PS^3, we construct a training set of 400 queries for each dataset by sampling at random the following aspects:

- between 0 and 8 columns as the group-by columns
- between 0 and 5 predicate clauses; each of which picks a column, an operator and a constant at random
- between 1 and 3 aggregates over one or more columns

We generate a held-out set of 100 test queries in a similar way. For `TPC-H*`, we include an additional test set of 10 TPC-H queries (§ 5.5.4). We ensure that there are no identical queries between the test and training sets and that there is substantial entropy in our choice of predicates, aggregates and grouping columns.

5.1.3 Methods of Comparison

All methods except for simple random sampling have access to feature vectors, and use the `selectivity_upper` feature to filter out partitions that do not satisfy the predicate before sampling. Recall that this filter has false positives but no false negatives. All methods have access to the same set of features. We report the average of 10 runs for methods that use random sampling.

Random Sampling. Partitions are sampled uniformly at random. Aggregates in the answer are scaled up by the sampling rate.

Random+Filter. Same as random sampling except that only partitions that pass the selectivity filter are sampled. This is only achievable with the use of summary statistics.

Learned Stratified Sampling (LSS). A baseline inspired by prior work on learned row-level stratified sampling [56]. We rank partitions by the model's prediction and perform stratification such that each strata covers partitions whose predictions fall into a consecutive range. We made three modifications to LSS to enable partition-level sampling: moving training from online to offline for I/O savings, changing inputs and outputs to operate on partitions instead of rows, and adopting a different stratification strategy. We include a detailed description of the modifications in Appendix C.1.

PS^3. A prototype that matches the description given so far. Unless otherwise specified, default parameter values for PS^3 in all experiments are $k = 4, \alpha = 2$ and up to a 10% sampling budget dedicated to outliers.

5.1.4 Error Metric

Similar to prior work [13, 16, 42], we report multiple accuracy metrics. It is possible, for example, for a method to have a small absolute error but miss all small groups and small aggregate values. We therefore consider all three metrics below for a complete picture.

Missed Groups. Percentage of groups in the true answer that are missed by the estimate.

Average Relative Error. The average of the relative error for each aggregate in each group. For missed groups, the relative error is counted as 1.

Absolute Error over True. The average absolute error value of an aggregate across groups divide by the average true value of the aggregate across groups, averaged over multiple aggregates.

5.2 Macro-benchmarks

We compare the performance of methods of interest under varying sampling budgets on four datasets (Figure 3). The closer the curve is to the bottom left, the better the results.

While the scale of the three error metrics is different, the ordering of the methods is relatively stable. Using the selectivity feature to filter out partitions that do not satisfy the predicate strictly improves the performance for all methods, except on datasets like `TPC-DS*` where most partitions pass the predicate. The modified LSS (green) clearly improves upon random sampling by leveraging the correlation between feature vectors and partition contribution, consistent with findings of prior work.

Overall, PS^3 consistently outperforms alternatives across datasets and error metrics. On our large scale experiment with the `TPC-H*` data, PS^3 achieves an average relative error of 1.5% with a 1% sampling rate. With a 1% same sampling rate, PS^3 improves the error achieved by 17.5× compared to random sampling, 10.8× compared to random sampling with filter and 3.6× compared to LSS (read from intersections between the baseline curves and a vertical line at 1% sampling rate). To achieve an average relative error of 1.5%, PS^3 reduces the fraction of data read by over 70× compared to random sampling, over 40× compared to random sampling with filter and 5× compared to LSS (read from the intersections between baseline curves and a horizontal line at 1.5% error rate). We observe similar trends on the three smaller datasets but the performance gap is smaller: PS^3 reduces the data read by 2.7× to 8.5× compared to simple random sampling to achieve $\leq$ 10% average relative error.

We additionally show that the fraction of data read is a reliable proxy for reductions in resources used, measured by total compute time. We evaluate example queries on the

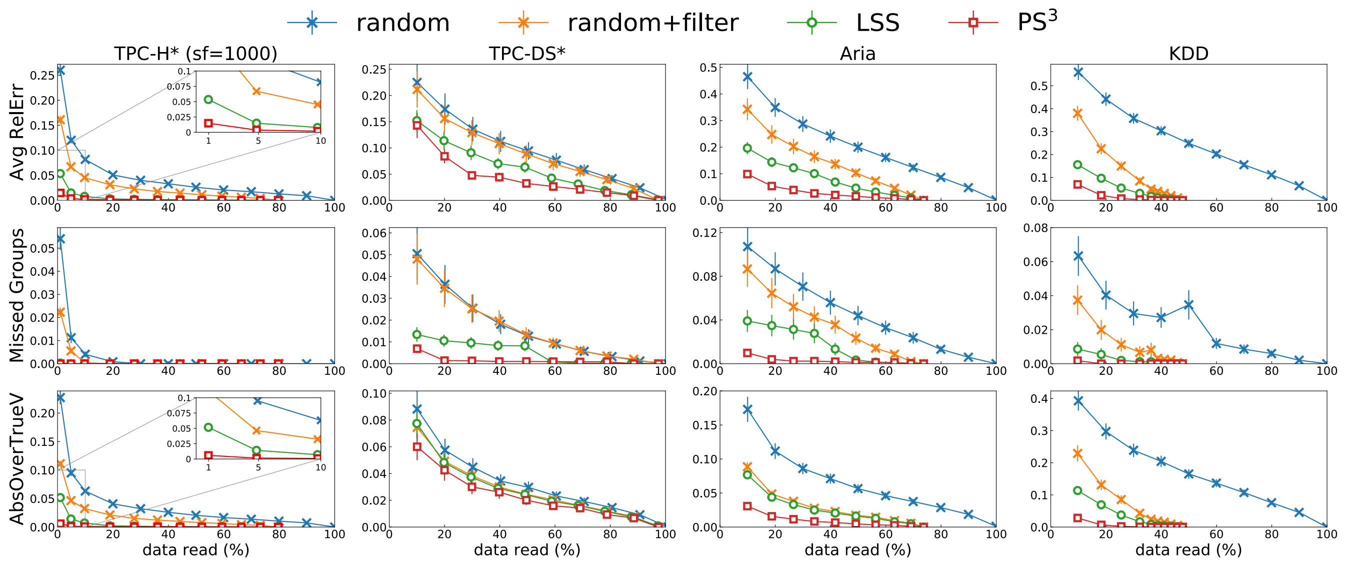


Figure 3: Comparison of error under varying sampling budget on four datasets, lower is better. PS^3 (red) consistently outperforms others across datasets and different error metrics.

Table 3: Average speedups for query latency and total compute time under difference sampling rates on the TPC-H* dataset.

	1%	5%	10%	100%
Query Latency	4.7×	1.6×	1.5×	-
Total Compute Time	105.3×	19.6×	11.4×	-

Table 4: Per partition storage overhead of the summary statistics (in KB) for each dataset.

Dataset	Total	Histogram	HH	AKMV	Measure
TPC-H*	84.25	9.52	13.26	55.31	6.16
TPC-DS*	103.49	10.51	4.67	81.45	6.86
Aria	18.38	1.42	0.81	15.19	0.97
KDD	12.00	2.19	0.82	5.29	3.70

TPC-H* dataset using SCOPE clusters [21, 59], Microsoft's main batch analytics platform, which consist of tens of thousands of nodes. Table 3 shows that reading 1%, 5% and 10% of the partitions results in a near linear speedup of 105.3×, 19.6×, 11.4× in the total compute time. Improvement of query latency however, is less than linear and depends on stragglers and other concurrent jobs on the cluster.

5.3 Overheads

We report the space overhead of storing summary statistics in Table 4. The statistics are computed for each column and therefore require a constant storage overhead per partition. The overheads range from 12KB to 103KB across the four datasets. The larger the partition size, the lower the relative storage overhead of the statistics. For example, with a partition size of 2.5GB, the storage overhead is below 0.003% for the TPC-H* dataset.

The AKMV sketch for estimating distinct values takes the most space compared to other sketches. If the number of distinct values in a column is larger than k (we use $k = 128$), the sketch has a fixed size; otherwise the sketch size is proportional to the number of distinct values. The

Table 5: Range of the average picker overhead across sampling budgets for each dataset (in milliseconds).

	Aria	KDD	TPC-DS*	TPC-H*
Total	89.9±4.7	106.4±4.9	219.6±4.7	1002.1±13.3
Clustering	24.1±5.0	58.0±2.2	148.0±5.4	802.4±12.8

KDD dataset, for example, has more columns but a smaller AKMV sketch size compared to the Aria dataset since a number of its columns are binary.

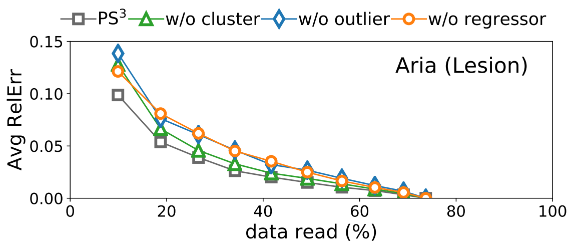
We also report the single-thread latency of the partition picker (Algorithm 1) in Table 5, measured on an Intel Xeon E5-2690 v4 CPU. Our prototype picker is implemented in Python using the XGBoost and Sklearn libraries. Overall, the overhead is a small fraction of the query time, ranging from 86.5ms to around 1s across datasets. In comparison, the average query takes tens of total computation hours on the TPC-H* dataset. As the number of partitions and the dimension of the feature vectors increase, the total overhead increases and the clustering component takes up an increasing proportion of the overhead. The overhead can be further reduced via optimization such as performing clustering in parallel across different importance groups.

5.4 Lesion Study

In this section, we take a closer look at individual components of the picker and their impact on the final performance, as well as the importance of partition features.

5.4.1 Picker Lesion Study

We inspect how the three components of the partition picker introduced in § 4 impact the final accuracy. To examine the degree to which a single component impacts the performance, we perform a lesion study where we remove each component from the picker while keeping the others enabled (Figure 4, top). To disable clustering (§ 4.2), we use random sampling to select samples. To disable identification of outlier partitions (§ 4.4), we take away the sam-



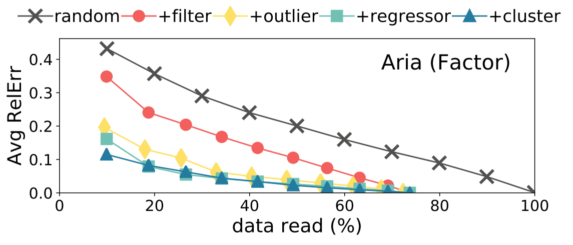


Figure 4: Lesion study and factor analysis on the Aria dataset. Each component of our system contributes meaningfully to the final accuracy. Results are similar on other datasets.

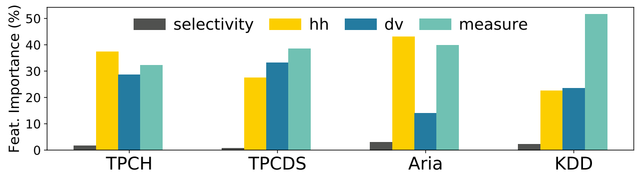


Figure 5: Feature importance for the regressors. The higher the percentage, the more important the statistics are to the regressor's accuracy.

pling budget dedicated to outliers. To disable the regressor (§ 4.3), we apply the same sampling rate to all partitions. The result shows that the final error increases when each component is disabled, illustrating that each component is necessary to achieve the best performance.

We additionally measure how the three components contribute to overall performance. Figure 4 (bottom) reports a factor analysis. We start from the simple random sampling baseline (random). Using `selectivity_upper` ≥ 0 as a filter (+filter) strictly improves the performance. Similar to the lesion study, we enable each component on top of the filter (not cumulative) while keeping others disabled. The results show that the identification of outlier partitions (+outlier) contributes the least value individually and the use of clustering (+cluster) contributes the most.

5.4.2 *Feature Importance*

We divide partition features into four categories based on the sketches used to generate them: selectivity, heavy hitter, distinct value and measures. We investigate the contribution of features in each component of PS^3. Filtering and outliers depend exclusively on histograms (selectivity) and heavy hitters (occurrence bitmap) respectively. We report the regressors' feature importance scores via the "gain" metric, which measures the improvement in accuracy brought by a feature to the branches it is on [9]. For each dataset, we report the total gain for features in each category as a percentage of the total importance score. The total score

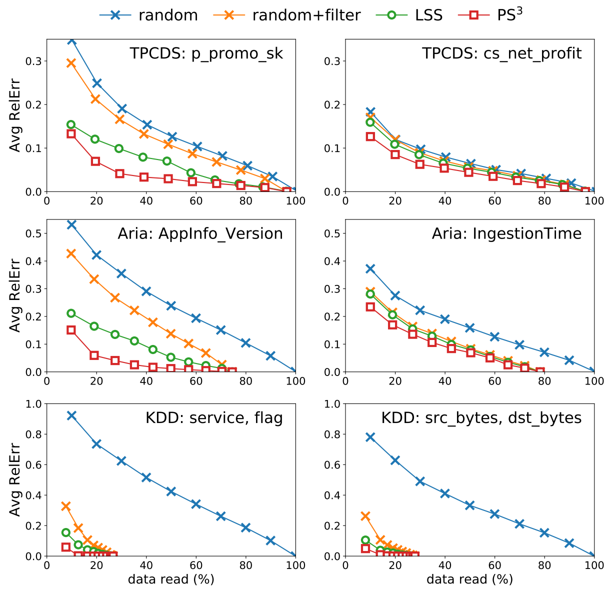


Figure 6: Our method consistently outperforms alternatives across datasets and data layouts.

is aggregated over all regressors used in the funnel. The larger the percentage, the more important the feature is to the final accuracy. We report the result in Figure 5.

Overall, all four types of features contribute to the regressor accuracy, but the relative importance varies across the datasets. Selectivity estimates, despite being less useful for regressors, are useful to filter out partitions that do not contain any rows satisfying the predicate.

5.5 Sensitivity Analysis

In this section, we evaluate the sensitivity of the system's performance to changes in setups and parameters.

5.5.1 *Effect of Data Layouts*

One of our design constraints is to be able to work with data in situ. To assess how PS^3 performs on different data layouts, we evaluate on two additional layouts for each dataset using the same training and testing query sets from experiments in § 5.2. Figure 6 summarizes the average relative error achieved under varying sampling budgets for the six combinations of datasets and data layouts.

PS^3 consistently outperforms alternatives across the board, but the sizes of the improvements vary across datasets and layouts. Overall, the more random/uniform the data layout is, the less room for improvement for importance-style sampling. For example in the `TPC-DS*` dataset, the layout sorted by column `cs_net_profit` is more uniform than the layout sorted by column `p_promo_sk`, since random sampling achieves a much smaller error under the same sampling budget in the former layout. LSS is only marginally better than random in the former layout, indicating a weak correlation between features and partition importance.

As a special case, we explicitly evaluate PS^3 on a random layout for the `TPC-H*` dataset with a scale factor of 1 (Figure 8, left). As expected, sampling partitions uniformly at random performs well on the random layout. PS^3 underperforms random sampling in this setting, but the performance

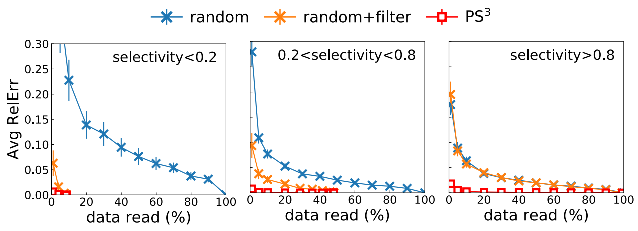


Figure 7: Performance breakdown by query selectivity on the `TPC-H*` dataset (sf=1000).

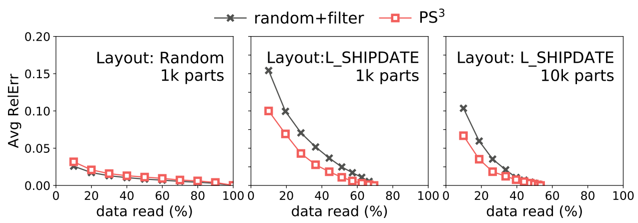


Figure 8: Comparison of `TPC-H*` (sf=1) results on different data layouts and total number of partitions.

difference is small. Realistically, we do not expect PS^3 to be used for random data layouts; users would have chosen random sampling were they paying the cost to maintain a random data layout [20].

5.5.2 *Effect of Query Selectivity*

We investigate how queries with different sensitivities benefit from PS^3. Figure 7 reports the error breakdown by query selectivity for random partition-level sampling and PS^3 on the `TPC-H*` dataset; other datasets show similar trends. Compared to naive random partition level sampling (blue), PS^3 offers more improvements for more selective queries (selectivity < 0.2), since the selectivity feature effectively filters out a large fraction of partitions that are irrelevant to the query. Compared to random partition level sampling with the selectivity filter (orange), PS^3 offers more improvements for non-selective queries (selectivity > 0.8), since they have larger errors at small sampling rates.

5.5.3 *Effect of Partition Count*

In this subsection, we investigate the impact of partition count on the final performance. We report results on the `TPC-H*` dataset (sf=1) with 1000 and 10,000 partitions in the middle and right plot of Figure 8. Compared to results on the same dataset with fewer partitions, the percentage of partitions that can be skipped increases with the increase of the number of partitions. In addition, as the partition count increases, the error achieved under the same sampling fraction becomes smaller. However, the overheads of PS^3 also increase with the number of partitions. Specifically, the storage overhead for per-partition statistics increases linearly with the number of partitions. The latency of the partition picker also increases with the partition count. Perhaps more concerning is the increase in I/O costs. The larger the partition count, the smaller the size of each partition. In the limit when each partition only contains one row, partition-level sampling is equivalent to row-level sampling, which is expensive to construct as discussed earlier (§ 1).

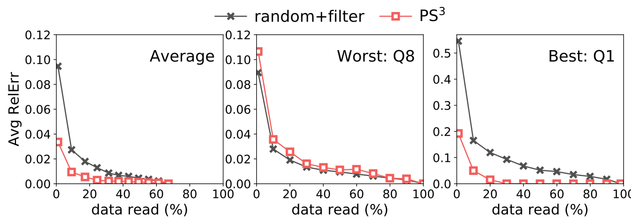


Figure 9: The average, worst and best results from a generalization test on unseen TPC-H queries (sf=1).

Table 6: AUC for different clustering algorithms; smaller is better.

	HAC(single)	HAC(ward)	KMeans
`TPCDS`	12.1	4.2	4.2
`Aria`	3.2	2.6	2.7
`KDD`	.71	.58	.55

5.5.4 *Generalization Test on TPC-H Queries*

To further assess the ability of the trained models to generalize to unseen queries, we test PS^3 trained on the randomly generated training queries with TPC-H schema (described in § 5.1.2) on 10 unseen TPC-H queries supported by our query scope[6]; the set of aggregate functions and group by columnsets are shared between the train and test set. We generate 20 random test queries for each TPC-H query template. We report the average, worst and best performances across the test queries on the `TPC-H*` dataset (sf=1000) in Figure 9. On average, PS^3 is still able to outperform uniform partition sampling, despite the larger domain gap between training and test set compared to experiments conducted in § 5.2. We report a detailed performance breakdown in Appendix C. Overall, we enable larger improvements on queries such as Q1, where a small number of partitions contain rare groups or outlying aggregate values. Our improvements are limited on Q8 which has a more complex aggregate and a nested query.

5.5.5 *Clustering Algorithm and Feature Selection*

We evaluate the effect of clustering algorithm choice and feature selection on the clustering performance.

We compare a bottom-up clustering algorithm (Hierarchical Agglomerative Clustering, or HAC) to a top-down algorithm (KMeans). For Agglomerative clustering, we compare two linkage metrics: the "single" linkage minimizes the minimum distances between all points of the two merged clusters, while the "ward" linkage minimizes the variances of two merged clusters. For each dataset, we evaluate the *average relative error* for estimating the query answer, and report the area under the error curve under different sampling budgets (Table 6). The smaller the area, the better the clustering performance.

HAC using the "ward" linkage metric and K-Means consistently produce similar results, suggesting that the clustering performance is not dependent on the choice of the clustering algorithm. The single linkage metric, however, produces worse clustering results compared to the "ward" linkage metric as well as the K-Means algorithm, especially on the `TPCDS` dataset.

[6]Q4 is excluded since it operates on the *orders* table.

6. RELATED WORK

In this section, we discuss related work in sampling-based AQP, data skipping, and partition-level sampling.

Sampling-based AQP. Sampling-based approximate query processing has been extensively studied, where query results are estimated from a carefully chosen subset of the data [25]. A number of techniques have been proposed to improve upon simple row-level random sampling, for example, by using auxiliary data structures such as an outlier index [23] or by biasing the samples towards small groups [12, 13]. Prior work has shown that, despite the improvements in sampling techniques, it is often difficult to construct a sample pool that offers good results for arbitrary queries given feasible storage budgets [44]. Instead of computing and storing samples apriori [14, 16, 22], our work makes sampling decisions exclusively during query optimization.

Prior works have used learning to improve the sampling efficiency for AQP. One line of work uses learning to model the dataset and reduce the number of samples required to answer queries [31]. Similarly, prior work tries to learn the underlying distribution that generates the dataset from queries, and relies on the learned model to reduce sample size over time [48]. Our work is closer to works that use learned models to improve the design of the sampling scheme. Recent work proposes a learned stratified sampling scheme wherein the model predictions are used as stratification criteria [56]. However, the work focuses on row-level samples and on count queries; we support a broader scope of queries with aggregates and group bys and work with partition-level samples. In the evaluation, we compare against a scheme inspired by learned stratified sampling.

Data Skipping. Our work is also closely related to prior works on data skipping which studied the problems of optimizing data layouts [54, 55, 58, 57] and indexing [40, 41, 49], improving data skipping given a query workload. Building on the observation that it is often difficult, if not impossible, to find a data layout that offers optimal data skipping for all queries, we instead choose to work with data in situ. Researchers and practitioners have also looked at ways to use statistics and metadata to prune partitions that are irrelevant to the query. The proposed approaches range from using simple statistics such as min and max to check for predicate ranges [3, 4], to deriving complex pruning rules for queries with joins [43]. Our work is directly inspired by this line of work and extends deterministic pruning to probabilistic partition selection.

Partition-level sampling. Researchers have long recognized the I/O benefits of partition-level sampling over row-level sampling [39, 50]. Partition-level samples have been used to build statistics such as histograms and distinct value estimates for query optimizers [24, 26]. Prior work has studied combining row-level and partition-level Bernoulli style sampling for `SUM, COUNT`, and `AVG` queries, in which one can adjust the overall sampling rate but each sample is treated equally [35]. Our work more closely resembles importance sampling where we sample more important partitions with higher probability.

Partition level sampling is also studied in the context of online aggregation (OLA) where query estimates can be progressively refined as more data gets processed, and users can stop the processing when the answer reaches target accuracy [28, 47, 51]. Classic work in OLA assume that tuples are processed in a random order, which often require random shuffling as an expensive processing step [20]. Our approach does not require random layout, and in fact, should not be used if the data layout is random. Prior work has also studied OLA over raw data, which requires an expensive tuple extraction step to process raw files [27]. PS^3 can work with data stored in any format as long as per-partition statistics are available and focuses on selecting fewer partitions instead of stopping processing early within a partition, since the most expensive operation for our setup is the I/O cost of reading the partition.

7. DISCUSSION AND FUTURE WORK

Our work shows promise as a first step towards using learning techniques to improve upon uniform partition-level sampling. We highlight a few important areas for future work below.

First, our system is designed mainly for read-only and append-only data stores, so the proposed set of sketches should be reconsidered if deletions and edits to data must be supported. Furthermore, the partition picker logic must be retrained when the summary statistics of partitions change in a substantial way.

Second, our work only considers generalization to unseen queries in the same workload on the same dataset and data layout. Although retraining can help generalize to unseen columns in the same dataset and layout, supporting broader forms of generalization such as to different data layouts is non-trivial and requires further attention.

Third, our work demonstrates empirical advantages to uniform partition-level sampling on several real-world datasets but provides no apriori error guarantees. Developing error guarantees and diagnostic procedures for failure cases will be of immediate value to practitioners.

8. CONCLUSION

We introduce PS^3, a system that leverages lightweight summary statistics to perform weighted partition selection in big-data clusters. We propose a set of sketches – measures, heavy hitters, distinct values, and histograms – to generate partition-level summary statistics that help assess partition similarity and importance. We show that our prototype PS^3 provides sizable speed ups compared to random partition selection with a small storage overhead.

9. ACKNOWLEDGEMENT

We thank Laurel Orr for her contributions to an early version of this project. We thank Surajit Chaudhuri and many members of the Stanford InfoLab for their valuable feedback. Kexin Rong, Peter Bailis and Philip Levis were supported in part by affiliate members and other supporters of the Stanford DAWN project—Ant Financial, Facebook, Google, Infosys, NEC, and VMware—as well as Toyota Research Institute, Northrop Grumman, Amazon Web Services, Cisco, the NSF under CAREER grant CNS-1651570 and the NSF under Grant No. CPS-1931750. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF. Toyota Research Institute ("TRI") provided funds to assist the authors with their research but this article solely reflects the opinions and conclusions of its authors and not TRI or any other Toyota entity.

10. REFERENCES

[1] https://bit.ly/2T8MsFj.

[2] Block Sampling in Hive. https://cwiki.apache.org/confluence/display/Hive/LanguageManual+Sampling. Accessed: 2020-2-12.

[3] Impala Partition Pruning. https://docs.cloudera.com/runtime/7.0.3/impala-reference/topics/impala-partition-pruning.html. Accessed: 2020-2-12.

[4] MySQL Partition Pruning. https://dev.mysql.com/doc/mysql-partitioning-excerpt/8.0/en/partitioning-pruning.html. Accessed: 2020-2-12.

[5] Oracle Database optimizer statistics. https://docs.oracle.com/en/database/oracle/oracle-database/18/tgsql/optimizer-statistics-concepts.htm. Accessed: 2020-2-12.

[6] PostgreSQL 9.5.21 TABLESAMPLE. https://www.postgresql.org/docs/9.5/sql-select.html. Accessed: 2020-2-12.

[7] Program for TPC-H Data Generation with Skew. https://www.microsoft.com/en-us/download/details.aspx?id=52430. Accessed: 2020-2-12.

[8] Snowflake SAMPLE / TABLESAMPLE. https://docs.snowflake.net/manuals/sql-reference/constructs/sample.html. Accessed: 2020-2-12.

[9] XGBoost Feature Importance. https://xgboost.readthedocs.io/en/latest/python/python_api.html. Accessed: 2020-2-12.

[10] *Probability Sampling from a Finite Universe*, chapter 1, pages 1–93. John Wiley & Sons, Ltd, 2009.

[11] F. Abuzaid, P. Kraft, S. Suri, E. Gan, E. Xu, A. Shenoy, A. Ananthanarayan, J. Sheu, E. Meijer, X. Wu, et al. Diff: a relational interface for large-scale data explanation. *PVLDB*, 12(4):419–432, 2018.

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

[13] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Acm Sigmod Record*, volume 29, pages 487–498. ACM, 2000.

[14] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

[15] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, et al. Spark sql: Relational data processing in spark. In *SIGMOD*, pages 1383–1394, 2015.

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

[17] S. D. Bay, D. Kibler, M. J. Pazzani, and P. Smyth. The UCI KDD archive of large data sets for data mining research and experimentation. *ACM SIGKDD explorations newsletter*, 2(2):81–85, 2000.

[18] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[19] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *SIGMOD*, pages 199–210, 2007.

[20] P. G. Brown and P. J. Haas. Techniques for warehousing of sample data. In *ICDE*, pages 6–6, 2006.

[21] R. Chaiken, B. Jenkins, P. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. Scope: Easy and e cient parallel processing of massive datasets. *PVLDB*, 1(2):1265–1276, 2008.

[22] S. Chaudhuri, G. Das, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *ACM SIGMOD Record*, volume 30, pages 295–306. ACM, 2001.

[23] S. Chaudhuri, G. Das, M. Datar, R. Motwani, and V. Narasayya. Overcoming limitations of sampling for aggregation queries. In *Proceedings 17th International Conference on Data Engineering*, pages 534–542. IEEE, 2001.

[24] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *SIGMOD*, pages 287–298, 2004.

[25] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *SIGMOD*, pages 511–519. ACM, 2017.

[26] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? *ACM SIGMOD Record*, 27(2):436–447, 1998.

[27] Y. Cheng, W. Zhao, and F. Rusu. Bi-level online aggregation on raw data. In *Proceedings of the 29th International Conference on Scientific and Statistical Database Management*, pages 1–12, 2017.

[28] X. Ci and X. Meng. An efficient block sampling strategy for online aggregation in the cloud. In *International Conference on Web-Age Information Management*, pages 362–373. Springer, 2015.

[29] G. Cormode, M. Garofalakis, P. J. Haas, C. Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

[30] G. Das, S. Chaudhuri, and U. Srivastava. Block-level sampling in statistics estimation, Oct. 6 2005. US Patent App. 10/814,382.

[31] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. *PVLDB*, 30:588–599, 2004.

[32] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *SIGMOD*, pages 679–694. ACM, 2016.

[33] E. Gan, P. Bailis, and M. Charikar. Coopstore: Optimizing precomputed summaries for aggregation. *PVLDB*, 13(11):2174–2187, 2020.

[34] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 29–43, 2003.

[35] P. J. Haas and C. König. A bi-level bernoulli scheme for database sampling. In *SIGMOD*, pages 275–286.

[36] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4):270–294, 2001.

[37] J. M. Hammersley and D. Handscomb. Percolation processes. In *Monte Carlo Methods*, pages 134–141. Springer, 1964.

[38] J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 171–182, 1997.

[39] W.-C. Hou and G. Ozsoyoglu. Statistical estimators for aggregate relational algebra queries. *ACM Transactions on Database Systems (TODS)*, 16(4):600–654, 1991.

[40] S. Idreos, M. L. Kersten, S. Manegold, et al. Database cracking. In *CIDR*, volume 7, pages 68–78, 2007.

[41] S. Idreos, S. Manegold, H. Kuno, and G. Graefe. Merging what's cracked, cracking what's merged: adaptive indexing in main-memory column-stores. *PVLDB*, 4(9):586–597, 2011.

[42] S. Kandula, K. Lee, S. Chaudhuri, and M. Friedman. Experiences with approximating queries in microsoft's production big-data clusters. *PVLDB*, 12(12):2131–2142, 2019.

[43] S. Kandula, L. Orr, and S. Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. *PVLDB*, 13(3):252–265, 2019.

[44] S. Kandula, A. Shanbhag, A. Vitorovic, M. Olma, R. Grandl, S. Chaudhuri, and B. Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *SIGMOD*, pages 631–646, 2016.

[45] E. Khalil, H. Dai, Y. Zhang, B. Dilkina, and L. Song. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, pages 6348–6358, 2017.

[46] G. S. Manku and R. Motwani. Approximate frequency counts over data streams. pages 346–357, 2002.

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

[48] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *SIGMOD*, pages 587–602, 2017.

[49] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *PVLDB*, 7(2):97–108, 2013.

[50] S. Seshadri and J. F. Naughton. Sampling issues in parallel database systems. In *International Conference on Extending Database Technology*, pages 328–343. Springer, 1992.

[51] Y. Shi, X. Meng, F. Wang, and Y. Gan. You can stop early with cola: online processing of aggregate queries in the cloud. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1223–1232. ACM, 2012.

[52] J. Shute, R. Vingralek, B. Samwel, B. Handy, C. Whipkey, E. Rollins, M. Oancea, K. Littlefield, D. Menestrina, S. Ellner, et al. F1: A distributed sql database that scales. 2013.

[53] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*. Ieee, 2010.

[54] L. Sun, M. J. Franklin, S. Krishnan, and R. S. Xin. Fine-grained partitioning for aggressive data skipping. In *SIGMOD*, pages 1115–1126, 2014.

[55] L. Sun, M. J. Franklin, J. Wang, and E. Wu. Skipping-oriented partitioning for columnar layouts. *PVLDB*, 10(4):421–432, 2016.

[56] B. Walenz, S. Sintos, S. Roy, and J. Yang. Learning to sample: Counting with complex queries. *PVLDB*, 13(3):390–402, 2019.

[57] Z. Yang, B. Chandramouli, C. Wang, J. Gehrke, Y. Li, U. F. Minhas, P.-r. Larson, D. Kossmann, and R. Acharya. Qd-tree: Learning data layouts for big data analytics. In *SIGMOD*, page 193–208, 2020.

[58] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *SIGMOD*, pages 17–30, 2015.

[59] J. Zhou, N. Bruno, M.-C. Wu, P.-A. Larson, R. Chaiken, and D. Shakib. Scope: parallel databases meet mapreduce. *PVLDB*, 21(5):611–636, 2012.

APPENDIX

A. DATA SCHEMA

A.1 TPC-H*

We provide the query used to denormalize the *lineitem* table in the TPC-H dataset below. This denormalized table can support 16 out of 22 queries in the TPC-H benchmark (Q1,3,4,5,6,7,8,9,10,12,14,15,17,18,19,21). We additionally include two derived columns L_YEAR and O_YEAR in the view in order to support group by clauses on these columns (Q7,8,9). Our generalization test (§ 5.5.4) includes the following 10 queries: Q1,5,6,7,8,9,12,14,17,18,19.

```
CREATE TABLE denorm AS
SELECT lineitem.*, customer.*, orders.*, part.*,
   partsupp.*, supplier.*, n1.*, n2.*, r1.*, r2.*,
   datepart(yy, o_orderdate) AS o_year,
   datepart(yy, l_shipdate) AS l_year
FROM lineitem JOIN partsupp ON ps_partkey = l_partkey
AND ps_suppkey = l_suppkey
   JOIN orders ON o_orderkey = l_orderkey
   JOIN part ON p_partkey = ps_partkey
   JOIN supplier ON s_suppkey = ps_suppkey
   JOIN customer ON c_custkey = o_custkey
   JOIN nation AS n1 ON n1.n_nationkey = c_nationkey
   JOIN nation AS n2 ON n2.n_nationkey = s_nationkey
   JOIN region AS r1 ON r1.r_regionkey = n1.n_regionkey
   JOIN region AS r2 ON r2.r_regionkey = n2.n_regionkey
```

A.2 TPC-DS*

We provide the query used to denormalize the *catalog_sales* table below. The joined dataset contains 4.3M rows, 21 numeric columns and 20 categorical columns.

```
CREATE TABLE denorm_cs AS
SELECT catalog_sales.*, cd.*, item.*, promo.*, date.*
FROM catalog_sales
   JOIN item ON cs_item_sk = i_item_sk
   JOIN promo ON cs_promo_sk = p_promo_sk
```

Table 7: Area under the curve for the average relative error of clustering under different sampling budgets for Hierarchical Agglomerative Clustering (HAC) and KMeans clustering; smaller is better.

	HAC (ward)	+feat sel	KMeans	+feat sel
TPCDS	4.2	3.8 (-9%)	4.2	3.8 (-8%)
Aria	2.6	2.3 (-14%)	2.7	2.3 (-15%)
KDD	.58	.55 (-5%)	.55	.54 (- .5%)

Algorithm 3 Feature Selection for Clustering

```
1: feats ← (selectivity, occurrence_bitmap,
        $\log(x)$, $\log^2(x)$, $\min(\log(x))$, $\max(\log(x))$,
        $\overline{x}, \overline{x^2}$, std, $\min(x)$, $\max(x)$,
        # hh, max hh, avg hh,
        # dv,avg dv, max dv, min dv, sum dv)
2: best ← []                ▷ Features excluded from clustering
3: for $i \leftarrow 1 \rightarrow 10$ do
4:     feats.shuffle()      ▷ Explore features in random order
5:     to_exclude ← []
6:     for $f \in$ feats do
7:         new ← [to_exclude]+[f]
8:         if ImproveCluster(to_exclude, new) then
9:             to_exclude ← new
10:        end if
11:    end for
12:    if ImproveCluster(best, to_exclude) then
13:        best ← to_exclude
14:    end if
15: end for
16: return best
```

JOIN date **ON** cs_sold_date_sk = d_date_sk
JOIN cd **ON** cs_ship_cdemo_sk = cd_demo_sk

A.3 Aria

`Aria` is a production service request log dataset at Microsoft that was also used in prior work [11, 33]. The data set contains the following columns: `records_received_count`, `records_tried_to_send_count`, `records_sent_count`, `olsize`, `ol_w`, `infl`, `TenantId`, `AppInfo_Version`, `UserInfo_TimeZone`, `DeviceInfo_NetworkType`, `PipelineInfo_IngestionTime`.

B. IMPLEMENTATION DETAILS

In this section, we provide additional implementation details for the partition picker.

B.1 Clustering

Normalization. Prior to clustering, we normalize the summary statistics to make sure that the euclidean distance is not dominated by any single statistic. We first apply a log transformation to reduce the overall skewness to all summary statistics except for selectivity estimates; for the selectivity estimates which are between 0 and 1, we use the cube root transformation instead. We then normalize each summary statistics by its average value in the training dataset. We choose the average instead of the max as the normalization factor since it is more robust to outliers. During test time, the statistics are normalized by their corresponding average values in the training dataset.

Failure Cases. As discussed in Section 4.2, clustering does not perform well when the predicate is highly selective. Although we can use the `selectivity_upper` feature as an upper bound for the true selectivity, in practice, we have seen that this upper bound could overestimated the true selectivity by over 10× for complex predicates (see Section 3.2). Therefore, we simply rely on the query semantics to estimate the complexity of the predicates. Specifically, if the predicate contains more than 10 clauses, we use random sampling instead of clustering to select sample partitions.

Feature Selection. We provide pseudo code for the feature selection procedure in Algorithm 3.

We report the features selected by the procedure on the four real-world datasets for experiments reported in § 5.2:

- `TPC-H*`: selectivity_upper, selectivity_lower, $\min(x)$, max hh, max dv, hh_bitmap
- `TPC-DS*`: $\log^2(x)$, $\overline{x}$, sum dv, hh_bitmap
- `Aria`: selectivity_indep, selectivity_max, $\min(\log(x))$, $\overline{x}$, $\max(x)$, avg hh, # dv
- `KDD`: selectivity_indep, $\overline{x^2}$, max dv

Only a small number of features are used in each dataset, but across datasets, all four types of features are represented. This again illustrates the need for all four sketches.

Finally, we measure the quantitative impact of the feature selection procedure on clustering performance in Table 7. Similar to the experiment in §5.5.5, we evaluate the *average relative error* for estimating the query answer using different clustering procedures, and compare the total area under the error curve for different sampling budgets. Overall, feature selection consistently improves clustering performance for both clustering methods, reducing the area from 0.5% to 15% across datasets.

B.2 Training

We use the `XGBoost` regressor as our base model and use the squared error as the loss function. Although our models are only used for binary classification, we train them as regressors instead of classifiers. This is to address the problem that the ratio of positive to negative examples are different for different queries. Consider a query which has one partition with rows that satisfy the predicate versus a query with 100 such partitions. Missing one positive example would have a much larger impact on the final accuracy for the first query compared to the second. While a classifier can only handle class imbalance globally, with a regressor, we can scale labels differently such that the positive examples weigh more in the first query. We provide pseudo code for the training set up in Algorithm 4.

C. ADDITIONAL RESULTS

C.1 Modified Learned Stratified Sampling

In this section, we present the three necessary modifications made to Learned Stratified Sampling [56] in detail:

- We move the training from online to offline, and use one trained model *per dataset and layout* instead of *per query*. LSS performs training inline for each query, using a fixed portion of the sampling budget as the training data. Training on random row-level samples may invalidate I/O gains and already require a full scan over data (§ 1). Instead, we train the model offline on

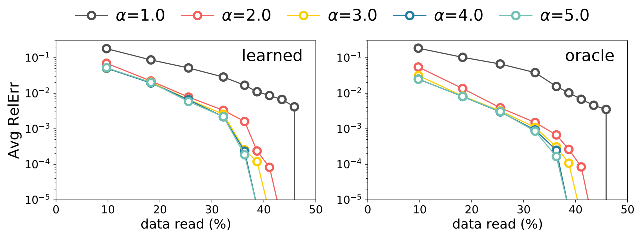


Figure 10: Impact of the sampling decay rate α on the KDD dataset. Larger α improves performance, but the marginal benefits decreases.

Algorithm 4 Training Label Generation

```
Input: threshold $t \in [0,1]$, partition count $n$, feature dimension $m$, query answer dimension $d$; for each input query $i$, partition features $F_i \in \mathbb{R}^{n\times m}$ and normalized query answers on each partition $A_i \in [0,1]^{n\times d}$
Output: X, Y
 1: $X \leftarrow [], Y \leftarrow []$
 2: for each $(F_i, A_i) \in$ training do            ▷ For each query
 3:     ans $\leftarrow \sum(A_i)$                    ▷ Ground truth query answer
 4:     for $j \leftarrow 1 \rightarrow n$ do
 5:         $y[j] \leftarrow \max(A_i[j]) > t$        ▷ Partition contribution
 6:     end for
 7:     positive $\leftarrow \sum y$
 8:     for $j \leftarrow 1 \rightarrow n$ do
 9:         if $y[j] == 1$ then
10:             $y[j] \leftarrow \sqrt{\frac{c}{positive}}$
11:         else
12:             $y[j] \leftarrow -\sqrt{\frac{c}{n-positive}}$
13:         end if
14:     end for
15:     $X.append(F_i)$
16:     $Y.append(y)$
17: end for
```

Table 8: Strata sizes for the modified LSS algorithm selected via exhaustive search.

	Sampling Budget (% data read)								
	10	20	30	40	50	60	70	80	90
`TPC-H*`	15	50	100	250	260	580	430	50	730
`TPC-DS*`	55	120	85	130	160	250	395	170	10
`Aria`	75	80	55	150	260	70	80	130	190
`KDD`	90	160	295	230	360	430	220	410	820

training queries sampled from the workload and use the same trained model for all test queries.

- We change the model's inputs and labels. LSS operates on rows, while we use partition features as inputs. LSS only considers count queries, so the label is either 0 or 1. To support aggregates and group bys, we use the partition contribution defined in § 4.3 as labels.
- We use different stratification strategies. Prior work analyzes optimal choices of strata boundaries for proportional allocation of samples, in which the sample size allocated to each stratum is proportional to its size. The analysis does not extend to our setup, so we use equi-width strata instead. To set the number of strata, we exhaustively sweep the strata sizes and select one that minimizes average relative error on the training set. We report the selected strata sizes in Table 8.

C.2 Effect of Sampling Rate

We investigate the extent to which applying different sampling rates affects the performance of learned importance style sampling. Recall that we tune the sampling rate via parameter α, which is the ratio of sampling rates between the i^{th} important and the $(i+1)^{th}$ important group. The larger α is, the more samples we allocate to the important groups. We report the results achieved under different αs for the `KDD` dataset (Figure 10, left). Overall the performance improves with the increase of α, but the marginal benefit decreases.

We repeat the experiment and replace the trained regressors with an oracle that has perfect precision and recall (Figure 10, right). This gives an upper bound of the improvements enabled by important-styled sampling. Compared to using learned models, the overall error decreases with the oracle, as expected. The performance gap between the learned and the oracle regressor increases with the increase of α. The comparison shows that the more accurate the regressor, the more benefits we get from using higher sampling rates for important groups. While we used a default value of $\alpha = 2$ across the experiments, it is possible to further fine-tune α for each dataset to improve the performance.

C.3 TPC-H Results

In this subsection, we report a detailed breakdown of the performances of PS^3 and random partition-level sampling on the TPC-H queries from the generalization test (§ 5.1.2). To support Q8 and Q14, we rewrite the `SUM` aggregate with a `CASE` condition as an aggregate over the predicate. In addition, PS^3 explicitly chooses to use random sampling instead of clustering to select samples for Q19, which has complex predicates consisting of 21 clauses (§ B.1). Our training queries are sampled randomly according to procedure described in § 5.1.2. We provide an example of a randomly generated query from the `TPC-H*` schema below:

```
SELECT N1_NAME,
       SUM(L_EXTENDEDPRICE * L_TAX)
FROM denorm
WHERE P_SIZE≥7 AND L_COMMITDATE≥"1997-09-29"
GROUPBY N1_NAME;
```

Overall, PS^3 significantly outperforms random partition selection on Q1, Q6 and Q7, and performs similarly to random partition selection on other queries. In particular, Q1, Q6 and Q7 all have a small number of partitions with either rare groups or outlying aggregate values. While PS^3 can identify such partitions via clustering and outlier detection, random partition selection can easily miss these important partitions especially when the sampling budget is limited.

D. VARIANCE ANALYSIS

D.1 Unbiased picker

Unbiased picker. We introduce an unbiased version of our proposed estimator that lends well to analysis. As described in § 4.2, the biased estimator picks an exemplar partition deterministically from a cluster given the median feature vector of the cluster, whereas the unbiased estimator

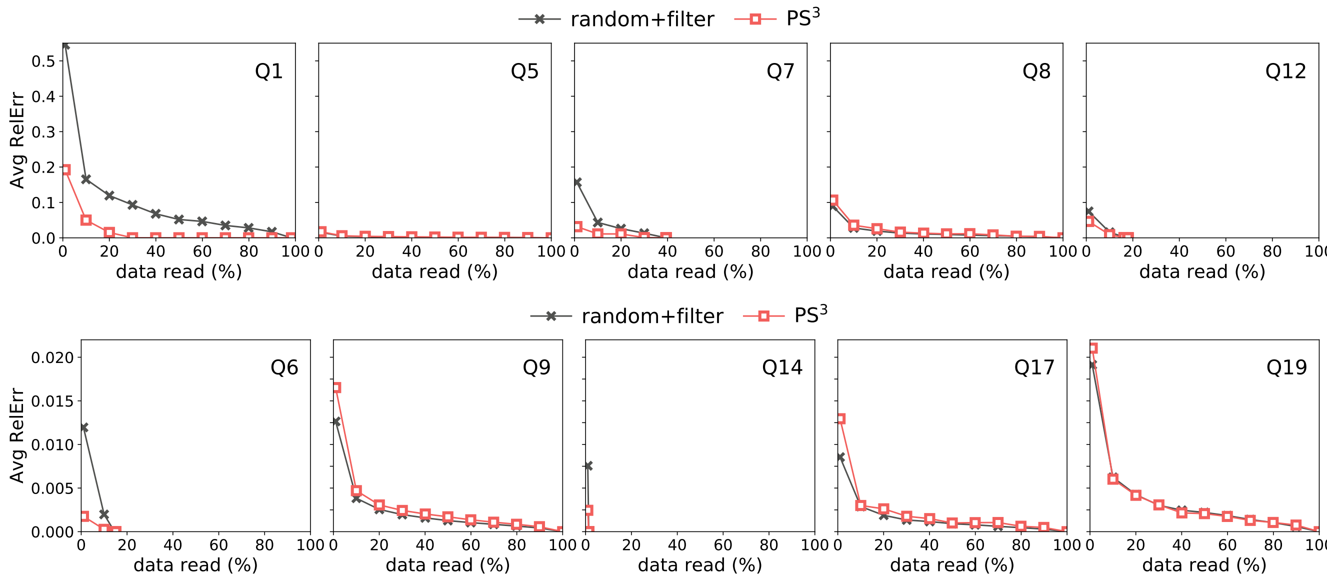


Figure 11: Detailed breakdown of results on TPC-H queries used in the generalization test (§ 5.1.2). Overall, PS^3 significantly outperforms random partition selection on Q1, Q6, Q7 and performs similarly to random partition selection on other queries.

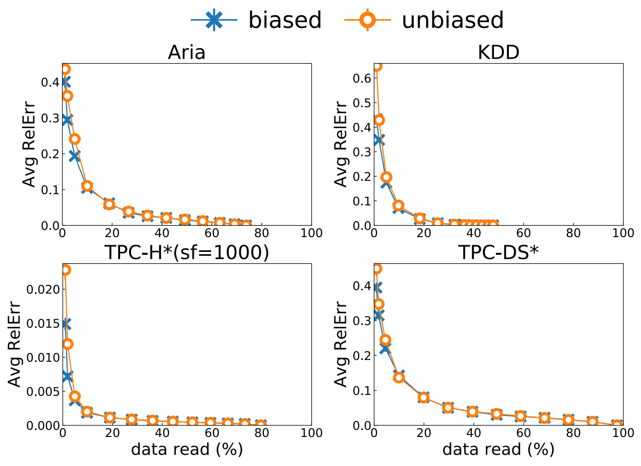


Figure 12: Empirical comparison of the bias and unbiased version of the estimator. The biased estimator tends to outperform the unbiased when the sampling fraction is small.

picks a cluster exemplar partition at random. We empirically compare the performances of the two estimators on four real-world datasets in Figure 12. For each test query, we run the unbiased estimator 10 times and compute the average error achieved to compare against the error achieved by the biased estimator.

Overall, Figure 12 shows that the biased estimator achieves smaller error compared to the unbiased version when the sampling fraction is small, and that there are no significant differences in accuracy between the two estimators otherwise. In addition, for a given query, the biased version of the estimator has *no variance*. Therefore, in use cases when the sampling budget is limited or when users prefer getting a deterministic answer for a given query, the biased version of the estimator might be preferred.

Analysis. Next, we analyze the unbiased version of the estimator using the framework of stratified sampling. Compared to a simple random sample of the same size, stratified sampling can produce an estimator with smaller variance if the elements within strata are homogeneous. In our case, each cluster is essentially a stratum; if clustering is effective, the partitions in a cluster are similar to each other, leading to a variance reduction.

Within each stratum, we perform simple random sampling without replacement (SRSWoR) to draw a sample of size 1; the variance formula for SRSWoR can be found in Chapter 2.5.2 of [29]. Note that since we only draw one sample from each cluster/stratum, in order to estimate variance of the stratum, we would need to evaluate *additional* partitions per stratum. Finally, the total variance of the unbiased estimator is the sum of the variances from each stratum.

When central limit theorem holds, the 95% confidence interval of an estimator Y is given by $\pm 1.96\sqrt{\sigma^2(Y)}$ [29], where $\sigma^2(Y)$ is the variance of the estimator described above.

D.2 Partition-level v.s. row-level sampling

In this subsection, we compare random partition level sampling to random row level sampling. We show that under the same sampling fraction, random partition level sampling has much larger variance than random row level sampling.

Set up. We start with a description of the setup. For a group G in the query, let y_i be the value of the aggregate function on partition i. Let π_i be the probability that partition i is included in the sample, π_{ij} be the probability that both partition i and j are in the sample, N be the total number of partitions and S be the set of sampled partitions.

We wish to estimate the total value of the aggregate function for group G on all partitions. For `SUM` and `COUNT` queries, the total value is $Y = \sum_{i=1}^{N} y_i$. If all partitions have positive sampling probability ($\pi_i > 0, \forall i$), an unbiased *Horvitz-Thompson* estimator for Y under Poisson sampling is:

$$\hat{Y} = \sum_{i \in S} \frac{y_i}{\pi_i}$$

The true variance of the estimator $\hat{Y}$ is:

$$\sigma^2(\hat{Y}) = \sum_{i,j=1}^{N} (\frac{\pi_{ij}}{\pi_i \pi_j} - 1) y_i y_j \tag{1}$$

However, since y_i is only available for partitions that are included in the sample, we can not evaluate the true variance using Eq 1 directly. Instead, we estimate the true variance using the sampled set of partitions S [29]:

$$\hat{\sigma}^2(\hat{Y}) = \sum_{i,j=1}^{N} (\frac{1}{\pi_i \pi_j} - \frac{1}{\pi_{ij}}) y_i y_j \tag{2}$$

If the second-order inclusion probability $\pi_{ij} > 0$ for all pairs of partitions i, j, Eq 2 is an unbiased estimator for Eq 1, the true variance of $\hat{Y}$ [10].

Analysis. For random partition level sampling, assume that each partition is selected in the sample with probability p. The expected size of S is Np. Since the partitions are sampled independently, $\pi_{ij} = \pi_i \pi_j$. Plug the inclusion probabilities in Eq 2, the estimator of the true variance is:

$$\hat{\sigma}^2(Y_{blk}) = \sum_{i \in S} (\frac{1}{p^2} - \frac{1}{p}) y_i^2 \tag{3}$$

Similarly, assume that each tuple is sampled with probability p. Let t_x be the total value that a tuple x contributes towards the aggregate for group G, and S_t be the set of sampled tuples. Following similar derivation as Eq 3, the estimator of the variance for random row level sampling is

$$\hat{\sigma}^2(T_{row}) = \sum_{x \in S_t} (\frac{1}{p^2} - \frac{1}{p}) t_x^2 \tag{4}$$

Note that y_i in Eq 3 is simply the sum of tuples in partition i. Let b_x be the partition that contains tuple x, then $y_i = \sum_{b_x = i} t_x$. Therefore,

$$y_i^2 = \sum_{b_x = i} t_x^2 + 2 \sum_{\substack{x<y, \\ b_x = b_y = i}} t_x t_y$$

Eq 3 can be rewritten as

$$\hat{\sigma}^2(T_{blk}) = \sum_{i \in S_t} (\frac{1}{p^2} - \frac{1}{p}) t_i^2 + 2 \sum_{\substack{i,j \in S_t, \\ i<j, b_i = b_j}} (\frac{1}{p^2} - \frac{1}{p}) t_i t_j \tag{5}$$

Comparing to random row-level sampling with the same sampling fraction p (Eq 4), random partition-level sampling has larger variance: Eq 5 includes an additional term that accounts for the variance contributed by tuples belonging to the same partition.