Approximate Partition Selection for Big-Data Workloads using Summary Statistics (Extended Version)

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

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 given a pre-partitioned dataset. 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 an up to 8.5× reduction in the number of partitions that must be read to achieve $\leq 10\%$ relative error compared to sampling the partitions uniformly at random, 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 these row-level samples [13, 17, 24].

When the data is contained in media that does not support random access (e.g., flat files in data lakes and columnar stores [53, 1]), constructing a row-level sample can be almost as expensive as scanning the entire dataset. For example, if data is partitioned into blocks with 100 rows, a 1% uniform row sample would in expectation require fetching 64% of the partitions; a 10% random row sample would require at least one row in every partition. 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].

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

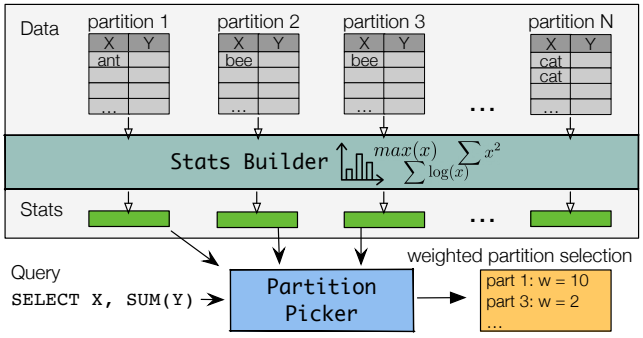

Figure 1: Our system PS^3 makes novel use of summary statistics to perform importance and similarity-aware sampling of partitions.

In contrast to row-level sampling, the I/O cost to construct a *block-level* sample can be proportional to the sampling fraction. In our example above, a 1% block sample requires reading 1% of the partitions. However, all or none of the rows in a partition are included in the sample, so correlation between rows (e.g., due to layout) can lead to a more inaccurate answer. We are especially interested in big data clusters, where data is partitioned and stored in chunks that can be tens to hundreds of megabytes, instead of disk blocks or pages which are typically a few kilobytes [33, 53][1]. Block-level sampling is already used in production due to its appealing performance: commercial databases create statistics using block samples [29, 6] and several Big Data stores allow sampling partitions of tables [3, 9, 7]. A key challenge remains in how to construct block-level samples that can answer a given query accurately – a uniformly random block-level sample is a representative row sample only if the rows are distributed randomly among blocks [25], which happens rarely in practice [21]. In addition, 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 [31] samples over partitions, which is required for 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

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

large-scale production query processing systems such as Spark [16], F1 [52], SCOPE [22] 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 network shuffles remain expensive and 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 precomputing and storing samples [23, 15, 17], which requires significant storage budgets to offer good approximations for a wide variety of queries [44, 26], PS^3 performs sampling exclusively during query optimization. Finally, similar to the query scope studied in prior work [47, 54, 15], PS^3 supports queries with `SUM, COUNT(*), AVG` aggregates, `GROUP BY` on columnsets with moderate distinctiveness, predicates that are conjunctions, disjunctions or negations over single-column clauses and key-foreign key joins.

To select blocks 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 compute and store. 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 that exhibit small space requirements, are easily maintained, and can be computed for each partition in one pass at ingest time. In terms of functionality, we look for statistics that are discriminative and rich 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. To keep the storage overhead low, we maintain single-column statistics, and the total size overhead scales with the number of blocks instead of with the dataset size. The resulting storage overhead can be orders of magnitudes smaller than approaches using auxiliary indices to reduce the cost of random access [31]. Although our primary focus is read-only and append-only data, in cases of updates, the statistics are also inexpensive to maintain compared to solutions based on precomputed aggregates [34, 48]. 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 the most to the query answer, we could improve the sampling efficiency by sampling the more important partitions more frequently. While it is challenging to manually design rules that relate summary statistics to the importance of a partition, with enough examples, a *learned* model can identify which summary statistics matter and how much a given partition influences a given query. Inspired by prior work which uses learning 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 rank partitions into several tiers of importance, and split the sampling budget in decreasing importance of tiers – more important tiers get a greater proportion of the sampling 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 case where new data partitions are continuously appended to the storage, models trained on old partitions can generalize to new partitions, assuming that the data distribution is stable.

In addition, we leverage the redundancy of 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 contents of the two partitions is expensive, we can use the summary statistics as a proxy for the similarity between partitions. In addition, we observe that datasets often 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. 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 [17, 24]. We again 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. We dedicate parts 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 8.5× 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 happen in practice [21]. 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 (§ 5.4.1).

Sampling on a single table. We argue that even for queries that involve joining multiple tables, it often suffices to perform sampling on only one table. First, in key — foreign key joins, fact tables are usually much larger in size compared to dimension tables. Sampling the fact table therefore, already gets us most of the performance gains. Second, prior work [44] has shown that in order to handle joins effectively, 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.

Generalization. Prior work 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 workload agnostic [38]. Our system falls in the middle of the spectrum, where we make weak assumptions about the query workload, similar to those in BlinkDB [15]. Specifically, we assume that we have access to the set of columns used in `GROUP BY`s and aggregate functions in the workload, and that this set is stable over time; predicates can take any form that fits under the scope defined in § 2.2. We assume that the workload consists of queries that contain an arbitrary combination of aggregates, group bys and predicates from the set. PS^3 is trained per data layout and workload, and generalizes to unseen queries in the workload.

However, we do not consider generalization to unseen data layouts or datasets. Since most summary statistics are computed per column, different datasets might not share any common summary statistics. Even for the same dataset, the importance of the same 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 provides no information in a random layout.

Confidence Intervals. Providing a-prior error guarantees is challenging for many AQP systems [26]. The problem is even more complex in the context of block-level sampling, where rows in a partition could be correlated. Combined with the usage pattern seen in practice [3, 9, 7, 6], we opt to work with a performance target rather than an accuracy target in this work. However, if desired, one could provide standard bootstrap-based error bounds by dedicating a small storage budget to a pool of random row samples of the dataset [58, 14]. This sample pool can be constructed at ingestion time with a single pass over the data, similar to other summary statistics. However, as a known caveat of using uniform random sampling, it does not provide tight confidence intervals when the predicate is selective.

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, groupbys and foreign key joins. Although we don't 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, extended version [2]).

- `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. 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 by default; to support derived attributes, we make a new column from the derived attribute and store its summary statistics. We do not support `GROUP BY` on columns with large cardinality since there is little practical gain from answering highly distinct queries over samples; one could either hardly perform any downsampling 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`. Equijoins between key and foreign-key columns are supported. 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 [47, 15, 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. The time complexity for constructing the sketches is linear in the number of rows, but the storage overhead is linear in the number of partitions. We describe the sketches used in detail in § 3.

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 data partition. Statistics on columns that are unused in the query are excluded.

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`. The picker generates a training query set from the specification, which consists of training queries that combine randomly sampled aggregate functions and group by columnsets (0 or 1) from the specification, as well as randomly generated predicates. For each query, we compute the summary statistics as well as the answer to the query on each partition, 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 generated by the statistics builder 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 a table in the query execution plan with a set of weighted partition choices.

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 if 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 might be of different sizes and magnitudes, we measure both absolute and relative error. In addition, we measure the percentage of groups that are in the true answer but is missed from the estimate.

3. PARTITION SUMMARY STATISTICS

The high-level insight of our approach is that we want to differentiate partitions according to their contribution to the query answer, and that the contribution is correlated with 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 if all else equal. We are unaware of prior work that uses partition level summary statistics for this purpose; our work explores the extent to which these statistics can improve block-level sampling beyond uniform. 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, 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 passes 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:

- **Measures** Minimum, maximum, as well as first and second moments are stored for each numeric column. For columns with positive values, 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 [20]. The sketch keeps track of the k minimum hashed values of a column as well as 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 a size up to 100.

Table 1: Time and space overheads per partition 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	$O(R_b)$	$O(\frac{1}{support})$

Table 1 summarizes the time complexity to construct the sketches and the space overhead to store them. 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 selectivity estimates that are computed for each query (see Table 2). We mask out features for columns that are unused in a query; for categorical columns where the measure based sketches don't apply, the corresponding features will also be zero. The schema of the feature vector is determined entirely by the schema of the table, so queries on the same table share the same set of feature vectors.

In summary, there are four types of features based on the underlying sketches that generate them: measures, heavy

Table 2: Overview of summary statistics and the sketched used to compute them. All statistics except for selectivity is computed for each column; selectivity is calculated for each query/predicate.

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

Algorithm 1 Partition Picker

Input: partition features F, sampling budget n, group bys columns *groups*, models *regrs*, decay rate α
Output: *part_choice* : $[(p_1, w_1), (p_2, w_2), ..., (p_n, w_n)]$
1: $outliers, remaining \leftarrow$ OUTLIER(F, *groups*)
2: $n_o \leftarrow outliers$.size()
3: $weights \leftarrow [1] * n_o$
4: *part_choice*.add(*outliers*, *weights*)
5: $buckets \leftarrow$ IMPORTANCEBUCKET(F, *remaining*, *regrs*)
6: $n_c =$ ALLOCATESAMPLES(*buckets*, $n - n_o$, α)
7: **for** $j \leftarrow 1, ..., buckets$.size() **do**
8: *part_choice*.add(CLUSTERING($F[buckets[i]]$, $n_c[i]$))
9: **end for**

hitters, distinct values and selectivity. Each type of feature reveals 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 (§ 5.3.2).

Most features can be extracted directly from the sketches; we elaborate on the two more involved ones below.

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 heavy hitters for a column by concatenating the heavy hitter dictionaries from each partition. For each partition, we compute a bitmap of size k, each bit representing whether the corresponding 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 given query predicate. The estimate supports predicates defined in our query scope (§ 2.2) and is derived using histograms over individual columns, with a few special cases. 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%'`). If two predicates are on the same column, we estimate the selectivity of the two predicates jointly; we assume independence otherwise. To compute selectivity over conjunctions and disjunctions, we estimate the selectivity for each individual predicate clause and store four related statistics:

1. `selectivity_upper`: For `AND`s, the upper bound on the selectivity is the min of the selectivity of individual clauses; for `OR`s, the upper bound is the min of 1 and the sum of selectivity estimates of individual clauses.
2. `selectivity_indep`: This feature computes the true predicate selectivity assuming columns are independent. For `AND`s, the feature is the product of the selectivity for each individual clauses; 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, `selecitivity_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 lower than 10%. By adding multiple measures related to selectivity, our hope is that they provide a more complete picture.

4. PARTITION PICKING WITH STATISTICS

In this section, we describe PS^3's partition picker component and how it makes novel use of the summary statistics 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 set of weighted partition choices consisting of 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 describes the entire procedure. We first identify outlier partitions with rare groups using procedure described in § 4.4. Each outlier partition has a weight of 1. We then use the trained models to sort the remaining partitions into buckets of different importance, using algorithm described in § 4.3. We allocate the remaining sampling budget across buckets such that the sampling rate decreases by a factor of α from the i^{th} important to the $(i+1)^{th}$ important bucket. Finally, given a sample size and a set of partitions in each bucket, we select samples via clustering using procedure described in § 4.2. An exemplar partition is selected from each cluster as a sample, and the weight of the exemplar equals the size of the cluster. We explain each component of the picker in detail in the following sections.

4.2 Sample via Clustering

We use feature vectors to compute a similarity score between partitions, which consequently enables us to choose dissimilar partitions as representative samples of the dataset. In fact, two identical partitions must have identical summary statistics. While the reverse does not hold, having summary statistics on multiple columns as well as multiple statistics for each column make it less likely that different partitions share 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 examplar 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 therefore given by $\tilde{A}. = \sum_{i=1}^{n} s_i A_i$.

Specifically, 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 ([2], 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. We only evaluate the input query on the exemplar partition for each cluster.

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 get clustered. Clustering is dynamic as well: the same partition can be in different clusters for different queries.

Feature Selection. We perform feature selection to improve the performance of clustering, which assumes that all features are equally important to partition similarity.

We perform a "leave-one-out" style test to determine a feature's usefulness for clustering. As an example, consider a table with two columns X, Y and two types of features min, max. We compare the clustering performance from using $\{min(X), max(X), min(Y), max(Y)\}$ as features to that from using only $\{max(X), max(Y)\}$ as features on the training set. If the latter gives a smaller error, we 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 to perform the "leave-one-out" test on. We include the pseudo code of the feature selection procedure in Appendix B of the extended version [2]. Our experiments show that feature selection consistently improves clustering performance across datasets (§ 5.4.5).

Limitations. Clustering implicitly assumes the existence of 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. Another failure case is when queries have highly selective predicates. 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. We fall back to random sampling in this case ([2], Appendix B).

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 answer and partitions that contribute less. The notion of contribution is useful since sampling the more important partitions more frequently could reduce the variance of the estimate [37].

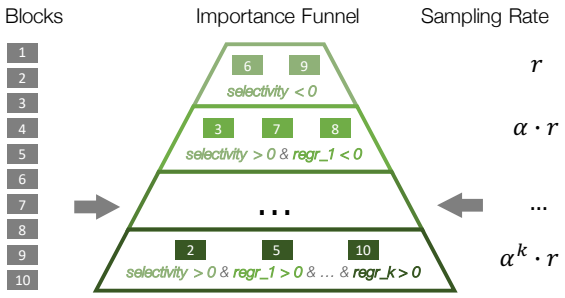

Figure 2: The trained regressors are used to sort input partitions into buckets of different importance. The sampling rate decrease by a factor of $\alpha > 1$ from the i^{th} important to the $(i+1)^{th}$ important bucket.

The feature vectors are useful in assessing partition importance. Consider the following example query: `SELECT SUM(X), Y FROM table WHERE Z > 1 GROUP BY Y`. The subset of partitions that would answer this query well should contain rows with disproportionately large values of X, contain many rows that satisfy the predicate and cover all distinct values of Y. Feature vectors provide supporting evidence for each of the desired characteristics: measures such as standard deviation and max reveal large values of X, selectivity reflects the effective fraction of the partition that is relevant to the query, and heavy hitter and distinct value statistics summarize the distribution of groups across partitions. However, it is challenging to manually quantify how much does each feature matter for each query. In the case of the example query, 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 variances and many rows matching the predicate.

While it is not obvious how to manually design rules that relate feature vectors to partition importance, a *learned* model may be able to do so automatically with enough examples. A natural design is to train a model on partition features and use the model's prediction as partition weights. This turns out to be a non-traditional regression problem. The ultimate 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, the optimal choice is discontinuous on n[2]. In addition, the decision depends jointly on the *set* of partitions chosen; the weight assigned to a 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 [45, 19] would require significantly more resources and we pursue a lightweight alternative instead.

Given the challenges to directly use a trained model for sampling, we leverage the learned model in a more controlled

[2]a small change in n can completely change the set of partitions to pick and the weights to assign to them

Algorithm 2 Using learned models as bucketing functions.

Input: partition features F

function ImportanceBucket($F, parts, regressors$)
  $buckets$.add(FilterByPredicate(F, parts))
  for $regr \in regressors$ **do**
    $to\_examine \leftarrow buckets[-1]$
    $to\_pick \leftarrow p \in to\_examine$ s.t. $regr(F[p]) > 0$
    $buckets[-1] = to\_examine$.difference(to_pick)
    $buckets$.add(to_pick)
  end for
  return $buckets$
end function

manner; similar observation was made for using learned models in a much simpler sampling setting with count queries [56]. Instead, we consider a classification problem in which we are only interested in the *relative importance* of partitions; the relative importance can be learned by picking up correlations between feature vectors and how much a partition contributes to the query answer. We propose a design based on this idea, which we describe below.

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. 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 denote the aggregates for group g on partition i 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 feature vector; when k is small, the training is a simplified version of the regression 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[3]. We use the `XGBoost` regressor as our base model, and provide additional details of the training in Appendix B of the extended version [2].

Testing. During test time, we run partitions through a funnel that utilizes the set of trained models as filters and sort partitions into different buckets 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 buckets, which help limit the impact of inaccurate models. We list the algorithm in 2. We start from all partitions with non zero `selectivity_upper` feature; as discussed in § 3.2, this filter has perfect recall but uncertain precision depending on the complexity of the predicates. We run the partitions through the first trained regressor, and move the ones that pass the regressor filter 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 split the sampling budget in decreasing importance of buckets – more important buckets get a greater proportion of the sampling budget. We implement a sampling rate that decays by a factor of $\alpha(>1)$ from the i^{th} important bucket to the $(i+1)^{th}$ important. We evaluate the impact of the decay rate α in the sensitivity analysis (§ 5.4). 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 decays rate of $\alpha = 2$ with $k = 4$ models works well across a range of datasets and layouts empirically; we leave the fine-tuning of the sampling rate to future work.

4.4 Outliers

Finally, we observe that datasets often exhibit significant skew in practice. Our production service request log dataset from Microsoft for example, contains records for 167 distinct application versions, but the most popular version accounts for almost half of the dataset. 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 [17, 24]. 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 cluster partitions with identical bitmap features for columns in the `GROUP BY` clause. We consider a bitmap cluster outlying if its size is small in both absolute and relative terms. For example, if there are a total of 100 such bitmap clusters and 10 partitions per cluster, we do not consider any as outlying although the absolute size of the clusters 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.

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 to 8.5× reduction of data read under 10% average relative error compared to uniform partition sampling, with storage overhead ranging from 12KB to 103KB per partition.

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

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 of the extended version [2].

TPC-H*. Data is generated from a Zipfian distribution with skewness of 1 and a scale factor of 1 [8]. We denormalize all tables against the *lineitems* table. The resulting table has 6M rows, 14 numeric columns and 31 categorical columns. Data is sorted by column *L_SHIPDATE*.

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

Aria. Production service request log at Microsoft [11, 32]. 10M rows, 7 numeric columns and 4 categorical columns. Data is sorted by categorical column *TenantId*.

KDD. KDD Cup'99 dataset on network intrusion detection [18]. 4.8M rows, 27 numeric columns and 14 categorical columns. Data is sorted by numeric column *count*.

By default we use a partition count of 1000, the smallest size from which partition eliminate becomes interesting. We evaluate the impact of the total number of partitions in the sensitivity analysis (§ 5.4.2). We also report results from additional data layouts for each dataset, including a random layout in the sensitivity analysis (§ 5.4.1).

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 zero to five predicate clauses; each of which picks a column, an operator and a constant at random
- between one to three aggregates over one or more columns

We include an example of a randomly generated query from the TPC-H* schema below:

SELECT O_ORDERPRIORITY,
SUM(L_EXTENDEDPRICE * L_DISCOUNT)
FROM denorm
WHERE R1_NAME = "EUROPE" **AND** P_SIZE $\geq$ 7
AND L_COMMITDATE $\geq$ "1997-09-29"
GROUPBY O_ORDERPRIORITY;

For each dataset, 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 (§ 5.4.3). We ensure that there are no identical queries between the test and training sets; that is, there is substantial entropy in our choice of predicates, aggregates and grouping columns.

5.1.3 Methods of Comparison

Table 3: Strata sizes for the modified LSS algorithm selected via exhaustive search.

	Sampling Budget (# partitions)								
	100	200	300	400	500	600	700	800	900
`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

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. The final estimate is scaled by the sampling rate.

Random+Filter. Same as random sampling except that only partitions that pass the selectivity filter are sampled.

Learned Stratified Sampling (LSS). A baseline inspired by prior work on learned row-level stratified sampling [56]. The core idea is leveraging learned models for stratification to reduce within-stratum variance. Specifically, 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 permit block-level sampling:

- We move the training from online to offline, and use one trained model *per dataset and layout* instead of *per query*. The original LSS performs training inline for each query, using a fixed portion of the sampling budget as 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 using data collected from multiple queries 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 (Table 3).

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 10% sampling budget dedicated to outliers.

5.1.4 Error Metric

Similar to prior work [13, 17, 42], we report multiple accuracy metrics.

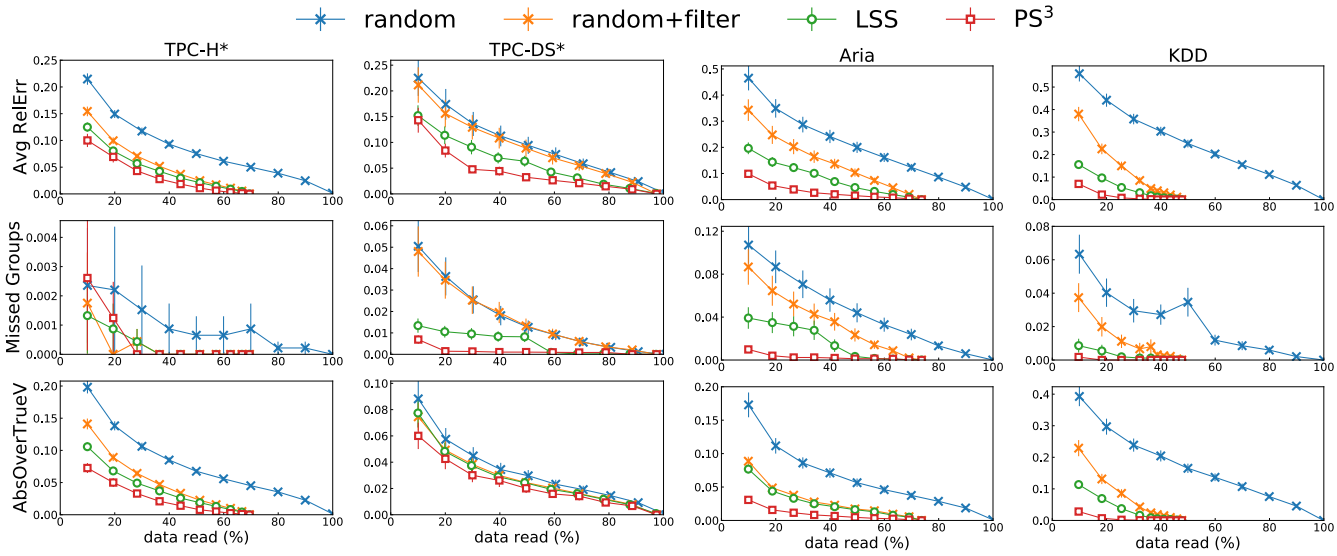

Figure 3: Comparison of error under varying sampling budget on four datasets, lower is better. PS^3 (red) consistently outperforms others across datasets and different error metrics.

Missed Groups. Percentage of groups in the true answer 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.

The last metric, which divides the absolute error over the true value, measures the magnitude of the error. However, it is possible to achieve a small absolute error and miss all small groups and all small aggregates. Therefore, we consider all three metrics for a complete picture.

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. Each column contains results for one dataset and each row reports a different error metric.

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 of all methods. However, the filter has minimal impact on datasets like `TPC-DS*`, where the filter almost always passes all partitions. The modified LSS (green) also clearly improves upon random sampling, consistent with findings of prior work. This improvement indicates that partition feature vectors are indeed correlated with the answer to a query on the partition.

Overall, PS^3 consistently outperforms alternatives across datasets and error metrics. To achieve $\leq 10\%$ average relative error on the test query set (first row), PS^3 reduces the fraction data read by 2.7 to 8.5× compared to simple random sampling, 1.9 to 5.0× compared to random sampling with filter and 1.5 to 3.5× compared to LSS. With a 20% sampling rate, PS^3 improves the average relative error by 2.1 to 19.9× compared to random sampling, 1.4 to 10.1× compared to random sampling with filter, and 1.2 to 4.3× compared to LSS across datasets.

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

We also 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. With a partition size of 100MB for example, the storage overhead would be below 0.1% 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, the sketch has a fixed size; otherwise the sketch size is proportional to the number of distinct values. The `KDD` dataset, for example, contains a number of binary columns, so it has a smaller AKMV sketch size compared to the `Aria` dataset, even though there are more columns. We inspect the usefulness of different features in the next section (§ 5.3.2).

5.3 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.

5.3.1 Picker Lesion Study

We inspect how the three components of the partition picker introduced in § 4 impact the final accuracy. To ex-

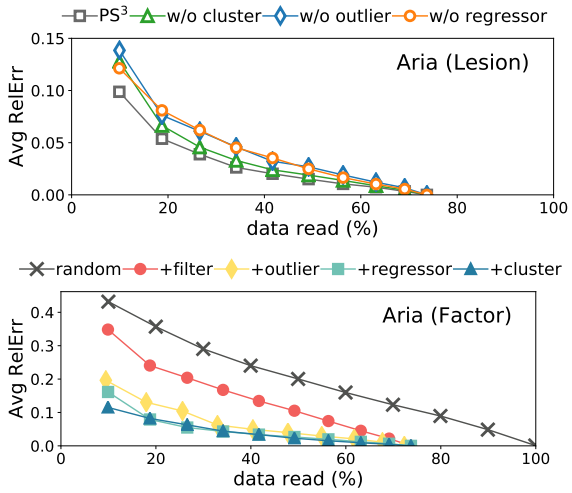

Figure 4: Lesion study and factor analysis on the Aria dataset. Each component of our system contributes meaningfully to the final accuracy.

amine 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 sampling 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 the identification of outlier partitions (+outlier), use of regressor for importance sampling (+regressor), and use of clustering to select samples (+cluster) individually on top of the filter (not cumulative). The results show that the outlier component contributes the least value individually and the clustering component contributes the most.

5.3.2 *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). For regressors, we estimate feature contribution using regressors' feature importance scores. We report the "gain" feature importance metric which measures the improvement in accuracy brought by a feature to the branches it is on [10]. For each dataset, we report the total importance score for features in each category as a percentage of the total importance score aggregated over all regressors used in the funnel; the larger the percentage, the more important the feature is to the

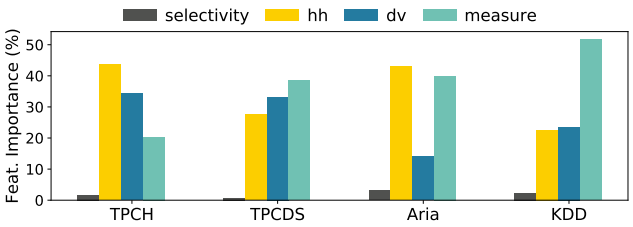

Figure 5: Feature importance for the regressors. The higher the percentage, the more important the statistics is to the regressor's accuracy.

final accuracy. We report the result in Figure 5.

Overall, all four types of features contribute to the regressor accuracy, but the relative importance of features vary across the datasets. In other words, no single feature dominates the regressor's performance. Selectivity estimates, despite being almost irrelevant for regressors, are useful in other components of PS^3.

For clustering, we report the features selected by the feature selection procedure (§ 4.2):

- `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.

5.4 Sensitivity Analysis

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

5.4.1 *Effect of Data Layouts*

One of our design constraints is to be able to work with data in situ. To access how PS^3 perform 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.

Similar to observations in § 5.2, our method is still able to consistently outperform alternatives across the board. However, 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. As an example, in dataset `TPC-DS*`, random sampling achieves much smaller average relative error in the layout sorted by column cs_net_profit compared to the layout sorted by column p_promo_sk; in the former layout, LSS is only marginally better than random sampling, 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 (Figure 7, 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 difference is small. Realistically, we do not expect PS^3 to be used for random data

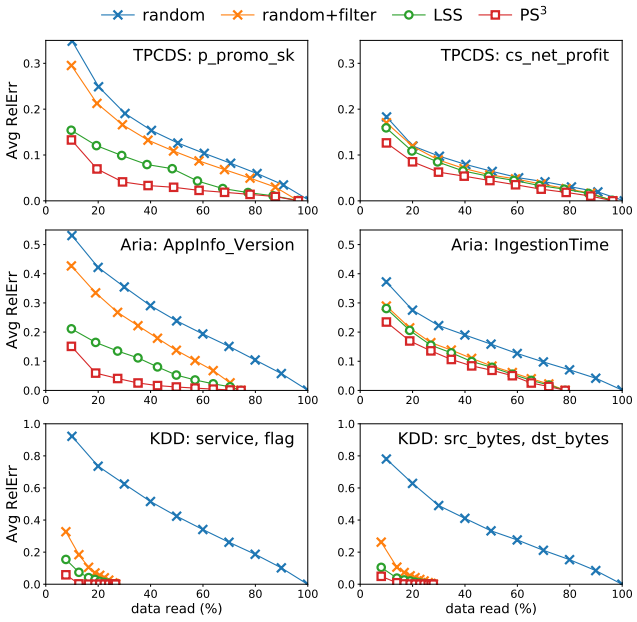

Figure 6: Our method consistently outperforms alternative across datasets and data layouts.

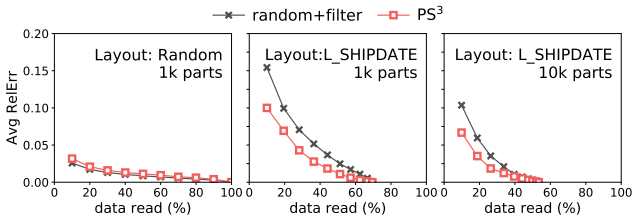

Figure 7: Comparison of TPC-H* results on different data layouts and total number of partitions.

layouts; users would have chosen random sampling were they paying the cost to maintain a random data layout [21].

5.4.2 *Effect of Partition Count*

Our experiments have used 1000 partitions thus far. We investigate the impact of partition count on the final performance, and report results on the TPC-H* dataset with 10,000 partitions in the rightmost plot of Figure 7. Compared to results on the same dataset with fewer partitions (Figure 7, middle), 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 cost of storing and constructing samples increases as the number of partitions increases. First, while the time complexity of computing statistics does not change with the increase in partition counts, the storage overhead increases linearly with the number of partitions. 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, block-level sampling is equivalent to row-level sampling, which is expensive to construct as discussed earlier (§ 1).

5.4.3 *Generalization Test on TPC-H Queries*

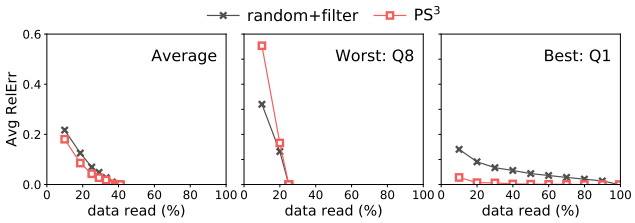

Figure 8: We report the average, worst and best results above on TPC-H queries.

To further assess the ability of the trained models to generalize unseen queries, we test PS^3 trained on the randomly generated query set using TPC-H schema (described in § 5.1.2) on 10 unseen TPC-H queries supported by our query scope[4]; the set of aggregate functions and group by columnsets are shared between the trained and test set. We generate 20 random queries from each TPC-H query template. We report the average, worst and best performances across the test queries in Figure 8. Overall, PS^3 outperforms uniform partition sampling on average, as well as on all queries except for Q8. Q8 has very selective predicates (only around 3% partitions contain rows that satisfy the predicate), a scenario in which clustering does not perform well, as discussed in § 4.2. We report a detailed performance break down by query in [2] (Appendix C).

5.4.4 *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, the ratio of sampling rates between the i^{th} important and the $(i+1)^{th}$ important bucket. The larger α is, the more samples we allocate to the important buckets. We report the results achieved under different αs for the KDD dataset (Figure 9, left). Overall the performance improves with the increase of α, but the marginal benefit decreases.

We repeat this experiment and replace the trained regressors with an oracle that has perfect precision and recall (Figure 9, right). This gives an upper bound of the improvements enabled by important-styled sampling. Compare 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 buckets. 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.

5.4.5 *Clustering Algorithm and Feature Selection*

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

We compare a bottom up clustering algorithm (Hierarchical Agglomerative Clustering) 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

[4]Q4 is excluded from the evaluation since it operates on the *orders* table instead of the *lineitems* table.

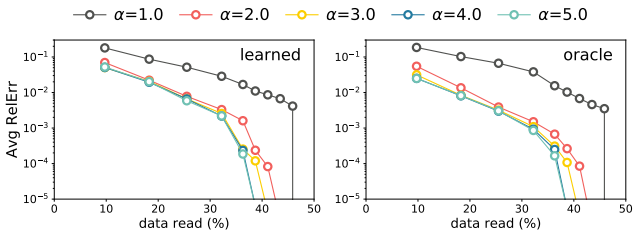

Figure 9: Impact of the sampling decay rate α on the KDD dataset. Larger α improves performance, but the marginal benefits decreases.

two merged clusters. For each dataset, we evaluate the *average relative error* achieved under different sampling budgets, and report the area under the curve (Table 5). 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. Feature selection consistently improves clustering performance for both clustering methods, reducing the area from 0.5% to 15% across datasets.

Table 5: Area under the curve for the average relative error of clustering under different sampling budgets; smaller is better.

	single	ward	+feat sel	KMeans	+feat sel
`TPCDS`	12.1	4.2	3.8 (-9%)	4.2	3.8 (-8%)
`Aria`	3.2	2.6	2.3 (-14%)	2.7	2.3 (-15%)
`KDD`	.71	.58	.55 (-5%)	.55	.54 (- .5%)

6. RELATED WORK

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

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 [26]. A number of techniques have been proposed to improve upon simple row-level random sampling, for example, by using auxiliary data structures like outlier index [24] or by introducing stratification to bias 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 [23, 15, 17], our work makes sampling decisions exclusively during query time.

We are also not the first to use learning to improve the sampling efficiency for AQP. There are two main approaches of combining learning and sampling. The first uses learning to model the underlying dataset; the more the model knows about the data, the less need for samples. For example, researchers have explored the idea of using a model to estimate sensor readings at the current time, which helps answer queries about the sensor network [30]. Similarly, prior work tries to learn the underlying distribution that generates the dataset over queries, and relies on knowledge of the learned model to reduce sample size and to improve result quality over time [47]. Our approach is closer to the second category, where the learned model is used to improve the design of the sampling scheme. A recent work proposes a learned stratified sampling scheme, in which the model predictions are used as stratification criteria [56]. However, the work and the analysis is focused on row-level samples and on count queries; we support a broader scope of queries with aggregates and group bys and work with block-level samples. In the evaluation, we compare against a scheme inspired by learned stratified sampling.

Block-level sampling. Researchers have long recognized the I/O benefits of block-level sampling over row-level sampling [39, 50]. Block-level samples have been used to build statistics such as histogram and distinct value estimation for query optimizers [27, 25]. [35] combines row-level and block-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. Our work more closely resembles importance sampling where we want to sample a block with probability proportional to the contribution to query answer. Block level sampling is also studied in the context of online aggregation where query estimates can be progressively refined as more data gets processed [51, 28]. However, the online aggregation setting typically requires data to be randomly shuffled as a preprocessing step, which can be expensive in practice [21]; the resulting data blocks can then be sampled at random (with or without stratification), and one can potentially stop processing early in the middle of a block. In contrast, our approach does not require random layout, and in fact, should not be used if the data layout is random.

Data Skipping. Our work is also closely related to the line of work in data skipping. Many prior work have studied the problem of optimizing data layouts [54, 55, 57] or indexing [40, 41, 49] to improve 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 as well as practitioners have also looked at ways to use statistics and metadata to prune partitions that are irrelevant to the query. A variety of approaches have been proposed, ranging from using simple statistics such as min and max to check for predicate ranges [4, 5], to deriving complex pruning rules beyond joins [43]. Our work is directly inspired by this line of work and extends deterministic pruning to probabilistic sampling.

7. CONCLUSION

In this paper, 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 summary statistics for each partition. We show that summary statistics can be used to assess partition similarity and importance and therefore improve weighted partition selection. Together, PS^3 provides sizable speed ups compared to random partition selection with minimum storage overhead.

8. REFERENCES

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

[2] Approximate Partition Selection for Big-Data Workloads using Summary Statistics (Extended Version). https://kexinrong.github.io/papers/ps3.pdf. Accessed: 2020-3-1.

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

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

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

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

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

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

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

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

[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. *Proceedings of the VLDB Endowment*, 12(4):419–432, 2018.

[12] S. Acharya, P. B. Gibbons, and V. Poosala. Aqua: A fast decision support system using approximate query answers. In *In Proc. of 25th Intl. Conf. on Very Large Data Bases*. Citeseer, 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, H. Milner, A. Kleiner, A. Talwalkar, M. Jordan, S. Madden, B. Mozafari, and I. Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 481–492, 2014.

[15] 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.

[16] 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 *Proceedings of the 2015 ACM SIGMOD international conference on management of data*, pages 1383–1394, 2015.

[17] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 539–550. ACM, 2003.

[18] 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.

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

[20] K. Beyer, P. J. Haas, B. Reinwald, Y. Sismanis, and R. Gemulla. On synopses for distinct-value estimation under multiset operations. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 199–210. ACM, 2007.

[21] P. G. Brown and P. J. Haas. Techniques for warehousing of sample data. In *22nd International Conference on Data Engineering (ICDE'06)*, pages 6–6. IEEE, 2006.

[22] 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. VLDB.

[23] 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.

[24] 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.

[25] S. Chaudhuri, G. Das, and U. Srivastava. Effective use of block-level sampling in statistics estimation. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 287–298. ACM, 2004.

[26] S. Chaudhuri, B. Ding, and S. Kandula. Approximate query processing: No silver bullet. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 511–519. ACM, 2017.

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

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

[30] A. Deshpande, C. Guestrin, S. R. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the Thirtieth international conference on Very large data*

bases-Volume 30, pages 588–599. VLDB Endowment, 2004.

[31] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample+ seek: Approximating aggregates with distribution precision guarantee. In *Proceedings of the 2016 International Conference on Management of Data*, pages 679–694. ACM, 2016.

[32] E. Gan, P. Bailis, and M. Charikar. Storyboard: Optimizing Precomputed Summaries for Aggregation, 2020.

[33] 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.

[34] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data mining and knowledge discovery*, 1(1):29–53, 1997.

[35] P. J. Haas and C. König. A bi-level bernoulli scheme for database sampling. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 275–286, New York, NY, USA, 2004. ACM.

[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. *Proceedings of the VLDB Endowment*, 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. *VLDB*, 12(12), August 2019.

[43] S. Kandula, L. Orr, and S. Chaudhuri. Pushing data-induced predicates through joins in big-data clusters. *Proceedings of the VLDB Endowment*, 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 *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 631–646, New York, NY, USA, 2016. ACM.

[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. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 346–357. Elsevier, 2002.

[47] Y. Park, A. S. Tajik, M. Cafarella, and B. Mozafari. Database learning: Toward a database that becomes smarter every time. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 587–602, New York, NY, USA, 2017. ACM.

[48] J. Peng, D. Zhang, J. Wang, and J. Pei. Aqp++: connecting approximate query processing with aggregate precomputation for interactive analytics. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1477–1492. ACM, 2018.

[49] F. M. Schuhknecht, A. Jindal, and J. Dittrich. The uncracked pieces in database cracking. *Proceedings of the VLDB Endowment*, 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 *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 1115–1126. ACM, 2014.

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

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

[57] E. Zamanian, C. Binnig, and A. Salama. Locality-aware partitioning in parallel database systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 17–30, 2015.

[58] K. Zeng, S. Gao, B. Mozafari, and C. Zaniolo. The analytical bootstrap: a new method for fast error estimation in approximate query processing. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 277–288, 2014.

APPENDIX

A. DATA SCHEMA

A.1 TPC-H*

We provide the exact query used to denormalize the *lineitems* 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 also 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 (§ 5.1.2) 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
  JOIN date **ON** cs_sold_date_sk = d_date_sk
  JOIN cd **ON** cs_ship_cdemo_sk = cd_demo_sk

A.3 Aria

This is a production service request log dataset at Microsoft. 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 as well as pseudo codes of algorithms.

B.1 Clustering

Before clustering, we normalize the summary statistics to ensure that the euclidean distance is not dominated by any single statistics. We first apply a log transformation to reduce the overall skewness to all summary statistics except for selectivity estimates. Given the selectivity estimates 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.

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 of 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 reply 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.

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

Algorithm 3 Feature Selection for Clustering

$feats \leftarrow$ (selectivity, 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)
$best = []$ ▷ Features excluded from clustering
for $i \leftarrow 1 \rightarrow 10$ **do**
  $feats$.shuffle() ▷ Random restart
  $excluded = []$
  repeat
    $prev\_excluded = excluded$
    for $f \in feats \setminus prev\_excluded$ **do**
      $new = [excluded] + [f]$
      if ImproveCluster($new, excluded$) **then**
        $excluded = new$
      end if
    end for
  until len($excluded$) = len($prev\_excluded$)
  if ImproveCluster($excluded, best$) **then**
    $best \leftarrow excluded$
  end if
end for
return $best$

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 within different queries. Consider a query which has 1 block with rows that satisfy the predicate versus a query with 100 such blocks. Missing one positive example would have a much larger impact on the final accuracy for the first query than the second. While a classifier can only handle class imbalance globally, with a regressor, we can scale labels differently such that the positive examples weight more in the first query. We provide pseudo code for the training set up in Algorithm 4.

C. ADDITIONAL RESULTS

In this section, we report a detailed breakdown of the performances of PS^3 and random partition selection on the TPC-H queries used in the generalization test (§ 5.1.2). To

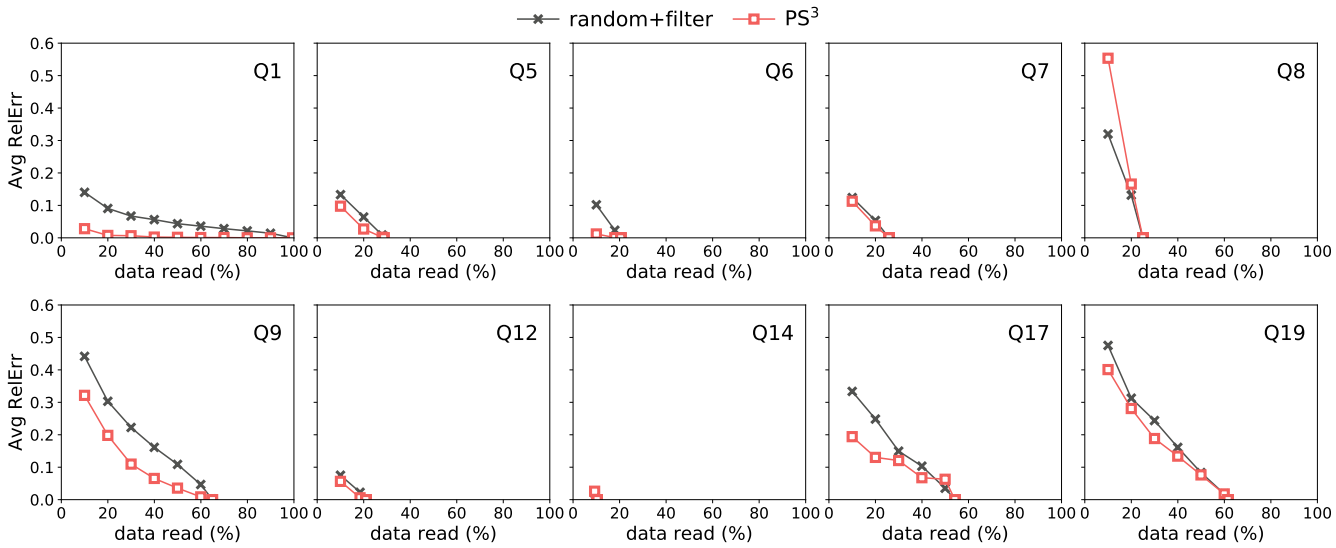

Figure 10: Detailed breakdown of results on TPC-H queries used in the generalization test (§ 5.1.2). Overall, PS^3 outperforms random partition selection on all queries except for Q8.

Algorithm 4 Training Label Generation

Input: threshold $t \in [0, 1]$, number of blocks n, block feature dimension m, query answer dimension d;

1: for each input query i: block features $F_i \in \mathbb{R}^{n \times m}$, normalized query answer on block i $A_i \in [0, 1]^{n \times d}$

Output: X, Y

2: $X \leftarrow []$
3: $Y \leftarrow []$
4: **for each** $(F_i, A_i) \in$ training **do** ▷ For each query
5: $ans = \sum(A_i)$ ▷ Ground truth query answer
6: $y \leftarrow [0, 0, ..., 0]$
7: **for** $j \leftarrow 1 \rightarrow n$ **do**
8: $y[j] \leftarrow \max(A_i[j]) > t$ ▷ Block contribution
9: **end for**
10: $nnz \leftarrow \sum y$
11: **for** $j \leftarrow 1 \rightarrow n$ **do**
12: **if** $y[j] == 1$ **then**
13: $y[j] \leftarrow \sqrt{\frac{c}{nnz}}$
14: **else**
15: $y[j] \leftarrow -\sqrt{\frac{c}{n-nnz}}$
16: **end if**
17: **end for**
18: $X.append(F_i)$
19: $Y.append(y)$
20: **end for**

support Q8 and Q14, we rewrite the aggregates as an aggregate over the predicate.

Overall, PS^3 outperforms random partition selection on all queries except for Q8. As discussed in § 5.1.2, Q8 has highly selective predicates but the `selectivity_upper` feature overestimates the true selectivity by over 7×, a scenario in which clustering does not perform well. In addition, Q8's predicate is semantically simple (only 5 clauses), so PS^3 does not fall back to random sampling. In comparison, 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). However, PS^3 is still able to outperform simple random sampling because it uses the learned models to group partitions into buckets of different importance, and assign higher sampling rates to the more important partitions. For Q5,6,7,12,14, the room of improvement is small, again limited by the selectivity of the predicate.