Efficiently Approximating Selectivity Functions using Low Overhead Regression Models

Anshuman Dutt, Chi Wang, Vivek Narasayya, Surajit Chaudhuri
Microsoft Research
{andut,wang.chi,viveknar,surajitc}@microsoft.com

ABSTRACT

Today's query optimizers use fast selectivity estimation techniques but are known to be susceptible to large estimation errors. Recent work on supervised learned models for selectivity estimation significantly improves accuracy while ensuring relatively low estimation overhead. However, these models impose significant model construction cost as they need large numbers of training examples and computing selectivity labels is costly for large datasets. We propose a novel model construction method that incrementally generates training data and uses approximate selectivity labels, that reduces total construction cost by an order of magnitude while preserving most of the accuracy gains. The proposed method is particularly attractive for model designs that are faster-to-train for a given number of training examples, but such models are known to support a limited class of query expressions. We broaden the applicability of such supervised models to the class of select-project-join query expressions with range predicates and IN clauses. Our extensive evaluation on synthetic benchmark and real-world queries shows that the 95th-percentile error of our proposed models is 10-100× better than traditional selectivity estimators. We also demonstrate significant gains in plan quality as a result of improved selectivity estimates.

PVLDB Reference Format:
Anshuman Dutt, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. Efficiently Approximating Selectivity Functions using Low Overhead Regression Models . *PVLDB*, 13(11): 2215-2228, 2020.
DOI: https://doi.org/10.14778/3407790.3407820

1. INTRODUCTION

Selectivity estimates are necessary inputs for a query optimizer, in order to identify a good execution plan [39]. A good selectivity estimator should provide accurate and fast estimates for a wide variety of intermediate query expressions at reasonable construction overhead [16]. Estimators in most database systems make use of limited statistics on the data, e.g., per-attribute histograms or small data samples on the base tables [41, 36, 35], to keep the query optimization overhead small [11]. However, these statistics are insufficient to capture correlations across query predicates and can produce inaccurate estimates for intermediate query expressions [11, 28]. Such inaccurate estimates often lead the optimizer to choose orders of magnitude slower plans [28]. While there is a huge literature on selectivity estimators (refer to survey [16]), accurate selectivity estimation with low overhead remains an unsolved problem.

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407820

1.1 Learned models for selectivity estimation

Recent works [25, 17, 46, 47] have shown that the supervised learned models may fit well into the low overhead expectation of query optimization, as they can provide better accuracy than traditional techniques with low estimation overhead. As a case in point, the regression model design proposed in [17] requires only tens of KB of memory and $\approx$ 100 μsec per estimation call, to deliver accurate estimates for correlated range filters on the base tables in the query. Their ability to adapt to the query workload, similar to self-tuning methods [10, 43], is also considered to be a significant advantage. However, constructing supervised models can take many hours [25, 46, 22], which is orders of magnitude slower than the traditional statistics collection methods.

To construct a supervised model for selectivity estimation of a given query expression, we need a set of example instances along with their true selectivities. Such training data is then used to train a regression model, which approximates the selectivity function captured by the given examples. It has been shown that the model training step is reasonably efficient - a few seconds with optimized libraries for gradient boosted trees [17] and several minutes for a simple neural network [46, 17]. The bottleneck lies in the generation of labeled training examples, where the overhead increases with (1) the number of query expressions to be supported by the model, (2) the number of training examples per query expression, and (3) the cost of generating true selectivity label for each training example. Recently, generation of labeled training examples has been highlighted as a major limitation of supervised models for selectivity estimation [22].

While past execution logs can provide labeled training examples without any explicit overhead, it is usually available only for a limited set of query expressions as highlighted in [47, 13]. For example, consider a query that joins 4 tables $\{T_1, T_2, T_3, T_4\}$ and executed using a right deep plan $(T_1 \bowtie (T_2 \bowtie (T_3 \bowtie T_4)))$. Execution logs can provide true selectivity labels for $(T_3 \bowtie T_4)$ and $(T_2 \bowtie (T_3 \bowtie T_4)$ but not for $(T_2 \bowtie T_3)$ or $(T_1 \bowtie T_3)$ and many other expressions.

1.2 Contributions

The focus of this work is to efficiently construct supervised selectivity estimation model for a given query expression without compromising accuracy. We present a novel model construction procedure that can reduce the training data collection overhead by reducing the number of training examples and the per-example label generation cost. Further, our analysis shows that faster-to-train models are better suited to reduce the total model construction cost for different query expressions with varying selectivity labeling cost. With this motivation, we build upon the low overhead regression models proposed in [17] to show that these simpler models can achieve accuracy similar to the more complex model design [25].

Efficient model construction. Recent proposals with supervised approach for selectivity estimation [25, 17, 46, 47] use tens of thousands of training examples for model training. They also empirically observed that similar accuracy could also be achieved with much smaller number of examples [17, 46] and approximated labels [17, 47]. We formalize these observations and propose a method to efficiently construct a supervised model with a target accuracy.

Automated determination of training data size Determining the required number of training examples is a challenging task [38, 23]. To avoid generating an unnecessarily large number of training examples, a potent approach is to use an iterative method that incrementally generates training examples. It can lead to significant savings, whenever a large enough test set is available to monitor the stopping criteria based on model accuracy [38, 23]. Also, if the model of choice is relatively slow to train, repetitive model training in each iteration can significantly increase the total time spent on model training. Overall, there is no existing iterative training method that can optimize the total model construction cost (model training cost and labeling cost) without compromising model accuracy.

We propose a novel iterative procedure that (1) supports early stopping for query expressions that need a small number of examples to train, by using cross-validation to avoid generation of a large set of test examples, (2) ensures robust monitoring of model accuracy by using confidence intervals on tail q-errors to suit the selectivity estimation context, and (3) optimizes the worst case total model construction cost by adapting the geometric step size to provide a constant-factor approximation guarantee. The optimal geometric step size is a function of the per-example *label_cost* for the query expression and the per-example *training_cost* for the chosen model training method. For instance, the constant-factor is ≈ 1.2, when *training_cost* is $100\times$ smaller than the *label_cost*.

Efficient approximation of training labels To reduce the absolute cost of training data generation in each iteration, we propose to use approximate selectivity labels for model training. To control the adverse impact on model accuracy, we use a small relative error threshold during this approximation. The key idea is that if the true selectivity has large value, it can be efficiently approximated using a relatively small uniform random sample, compared to the case when true selectivity is small[1]. The challenge lies in identifying appropriate sample size for different examples. We present an algorithm that takes a set of unlabeled training examples and an error threshold as input, and uses uniform random data samples of increasing sizes to progressively refine the probabilistic estimate of the selectivity labels, until sufficiently accurate labels are determined.

Extended applicability of regression models. With reduced *label_cost* due to approximation, *faster-to-train* models proposed in [17], such as tree-ensembles, become more attractive as they can ensure that the total model construction overhead is closer to its lower bound, i.e., the cost of generating only the required number of training examples. But these models were evaluated only for single table range predicate [17]. The second contribution of this work is to demonstrate that low overhead regression models [17] can be extended to support selectivity estimation for select-project-join queries with multiple categorical (IN) and range filters on the base tables, which is an important subclass of queries. We also discuss how these models can support other filter types by leveraging run-time features, i.e., *sample-bitmap* [25]. Finally, our model design choices focus on reducing estimation overhead.

Using models for join estimates Regression models for single tables [17] can be adapted for joins by following the same approach as statistics on views [9, 19], by treating a materialized view for the join as a base table. The materialized join can be discarded after constructing the required model. However, the cost of materialization adds up to the model construction cost. We empirically found that a reasonable approximation of the join selectivity function can be learned using a large enough sample of the join, which can be collected relatively efficiently without materializing the entire join result using existing techniques [18, 45, 50].

Overall, our models can typically deliver accuracy comparable to join-samples by using only compile-time information similar to histograms. While we need to create a dedicated model for each query expression (refer to Section 5.5 for exceptions) rather than a single global model as in [25], the total memory footprint does not blow up quickly as individual models are small.

Extensive evaluation and plan improvements. We present an extensive evaluation with 42 query expressions across 3 different datasets using queries with joins up to 5 tables with filter predicates on multiple base tables. We show that regression models with only 16KB memory can deliver 10-100$\times$ better 95^{th} percentile error values compared to traditional techniques such as histograms, statistics on views, and join-samples. Surprisingly, our 16 KB regression models delivered accuracy comparable to custom design supervised models [25] with much larger number of model parameters. Our model construction improvements bring $10\times$ or more savings for a large fraction of expressions compared to existing training method. Finally, we also evaluate improvement in quality of plans when estimates produced by our trained model are injected during query optimization. A sample experiment consisting of 500 test query instances with a fixed query template show that injected estimates bring improvement similar to injecting true selectivities. Overall, 30% queries improved by a factor of at least 1.2 and 10% of the instances improved by a factor of $10\times$ or more.

[1]We highlight the similarity and differences w.r.t. approximate query processing literature [7, 6] in Section 4.2.

Organization. We start with a quick review of selectivity estimators in existing optimizers in Section 2 and formally define our problem and evaluation metrics in Section 3. The algorithmic improvements to reduce model construction overhead are discussed in Section 4, followed by details on model design and feature encodings in Section 5. Section 6 provides experimental setup, and summarizes our empirical findings. Finally, we review related literature in Section 7 and conclude in Section 8.

2. BACKGROUND

The quality of the execution plan chosen by the query optimizer heavily depends on the output size estimates at intermediate stages of the plan [28], often termed as *selectivity* estimates[2]. In this section, we review the important characteristics of a good selectivity estimator and the state-of-the-art estimators in existing industrial optimizers.

2.1 Criteria for a good selectivity estimator

A selectivity estimator needs to be evaluated on the following criteria: (1) accuracy, (2) space and time efficiency, (3) construction overhead, and (4) range of applicability [16]. Estimation accuracy is critical for selecting good quality plans and small estimation time is important to keep check on optimization overhead [11]. Typically, each estimator requires a metadata structure (e.g. histogram, sample rows, regression model) that decides the memory footprint of using the estimator. The issue of space efficiency is further emphasized by the fact that the estimator may need a separate structure for each query expression. While construction of these structures is offline, i.e., it does not impact the critical path of query execution, it is an important consideration due to the large number of query expressions and handling updates to the database. Finally, range of applicability is another very important criteria for a good selectivity estimator. It identifies the subclass of query expressions that the estimator can support and is important because estimates are required for a large variety of query expressions that involve a mix of various relational operators such as select (filters), joins, group by, union etc.

2.2 Existing selectivity estimators

Most existing optimizers [36, 41, 2] use table-level statistics, e.g., row count, and attribute-level statistics that include null count, distinct count, and histograms or data samples on the base tables [36, 35]. Selectivity estimates for intermediate expressions, e.g., combination of filters, joins etc., are typically derived from table-level and attribute-level statistics using a set of assumptions including attribute value independence (AVI), join containment, and uniform value distribution etc. [41]. These techniques satisfy most of the requirements of a good estimator such as small space and time overhead, low construction overhead, and applicable for most query expressions in select-project-join queries[3]. But they are prone to large estimation errors [28] when correlations are present among various query predicates, e.g., between two filter predicates or between filter and join predicates. Per-attribute histograms can also be built over a materialized view [9, 8, 19] to capture join-correlation at the expense of extra construction overhead, they still do not capture correlation between view attributes.

There have been many proposals in research literature including multi-dimensional histograms, and other statistical structures, as reviewed in Section 7. Most of them either have large construction or space/time overhead, or have limited applicability. Sampling based techniques [45, 27, 5, 21, 15, 49, 18] can deliver good accuracy, reasonable construction overhead and have wide range of applicability, their space/time overhead do not fit the low overhead expectation of query optimization.

In this work, we explore simple regression models for selectivity estimation [17]. Such models are attractive because they can capture data correlations to provide significantly better accuracy compared to traditional techniques, while keeping low estimation overhead [47, 17, 25, 46]. We aim to improve their usability by significantly reducing their construction overhead and broadening their applicability.

3. PROBLEM DESCRIPTION

Selectivity estimation is very challenging in its full generality. In this work, we focus on select-project-join queries with simple filters such as range and categorical filters (IN clause), which is an important subclass of queries. Observe that this subclass is similar to that of statistics on views [8, 9, 19]. Specifically, we focus on estimation accuracy, and overhead (estimation and construction) for a given query expression of the above mentioned class.

3.1 Notations

Consider a relation T with attributes A_1,A_2,..., A_d, where k^{th} attribute A_k is either a numerical attribute with domain $[min_k, max_k]$ or a categorical attribute with finite domain $catg^k = \{catg_1^k, catg_2^k, \ldots, catg_N^k\}$. Observe that, the relation T here can be a base table in the database or a relation that represents the result of a join between two or more base tables. A range filter predicate on numerical attribute is of the form $(lb_k \leq A_1 \leq ub_k)$ where $lb_k \geq min_k$ and $ub_k \leq max_k$. An IN $filter$ predicate on categorical attribute is a non-empty subset of $catg^k$.

Let q represent a query on relation T consisting of *conjunction* of range and categorical predicates. In this representation, if the query does not contain a predicate on some numerical attribute A_k, then it is treated as $min_k \leq A_k \leq max_k$. Similarly, if the query does not contain a predicate on some categorical attribute then it is treated as the entire domain $catg^k$. For instance, if relation T has a numerical attribute A_1 with domain [0,100] and categorical attribute A_2 with domain $\{catg_1^2, \ldots, catg_{10}^2\}$. Then, $(10 \leq A_1 \leq 20) \wedge A_2$ IN $(catg_1^2, catg_2^2)$ is an example query.

We define *actual selectivity* of q as the number of rows in relation T that satisfy *all* predicates in the query q, and denote it with $act(q)$. Similarly, we use $est(q)$ to denote the estimated selectivity for query q. Let Q represent a set of m queries $Q = \{q_1, \ldots, q_m\}$. And S represents the labeled version of query set Q, i.e., $S = \{(q_1 : act(q_1)), \ldots, (q_m : act(q_n))\}$, with actual selectivity as the label, e.g. $\{(10 \leq A_1 \leq 20 \wedge A_2$ IN $(catg_1^2, catg_2^2) : 5000), (70 \leq A_1 \leq 85 \wedge A_2$ IN $(catg_4^2, catg_9^2) : 300), \ldots\}$.

Observe that, supervised models are typically studied in a training-testing regime. That is, a model M trained using labeled query set S is expected to produce reasonably accurate estimates for queries in S_{test}, when both S and S_{test}

[2]We use the term "selectivity fraction" to denote the fraction of output rows out of the maximum output size.

[3]Assuming simple filters of the form *column* $\langle op \rangle$ *constant*

have queries sampled from the same distribution. In this work, we use a random query distribution to generate the query set Q for a given relation T (details in Section 6.1).

Relevance of the past execution logs In practice, past execution logs can be leveraged to construct the required labeled query examples, even when the logs may not provide sufficient examples to be directly used for model training. For example, it can help identify the set of query expressions that need to be supported using the supervised models, depending on the usage frequency in the workload, and the error-profile with the default estimation method. It can also guide the parameters that define the space of queries for the training query generation method with information such as the set of relevant filter attributes and their sub-domains. For instance, the range of length of IN clause and the subset of categories can be quite useful in generating training examples relevant to the workload.

3.2 Problem Statement

We consider the problem of efficiently constructing a model M from a given unlabeled query set Q, such that it satisfies a specified accuracy target when evaluated on S_{test}. Observe that, the model construction cost includes the time required to generate selectivity labels for examples in Q as well as the time to train the model M. Our goal is to reduce the total model construction cost compared to one-shot model training with a fixed large number of training examples, without compromising accuracy.

Observe that the procedure may not provide required accuracy when the given query set size m of training examples, model input feature set, or the model size is insufficient - we discuss such practical considerations in Section 4.3.

3.3 Evaluation metrics

To evaluate training efficiency of our solution, we report the number of training examples used for training, denoted with m', out of the maximum number of examples m and also report total CPU-time spent in constructing model M. To evaluate the estimation accuracy of model M, we use *q-error* as the metric, a widely used metric in selectivity estimation context [32, 28, 17, 25, 34]. Each query $q_i \in S_{test}$ has a q-error $e_i = \max\left(\frac{est(q_i)}{act(q_i)}, \frac{act(q_i)}{est(q_i)}\right)$. To make the q-error well-defined, we assume that $act(q) \geq 1$ and $est(q) \geq 1$. To avoid artificially large values of q-error, we ignore errors due to queries that have $act(q_i) \leq \Delta$ as well as $est(q_i) \leq \Delta$ [31], for a small constant Δ. To evaluate aggregate accuracy of M over S_{test}, we either plot entire q-error distribution or use 95^{th} percentile of q-error values.

We include estimation time as well as model size (in KB) for constructed models in our evaluation. Finally, we also report the impact on plan quality, when we inject model estimated selectivities for the query expressions involved during the query optimization.

4. EFFICIENT MODEL CONSTRUCTION

Supervised models for selectivity estimation are quite attractive for multiple reasons: (1) low overhead estimation method as preferred during query optimization [27, 11], (2) ability to adapt to recent query workload patterns, similar to self-tuning methods [43, 10], and (3) potential to deliver selectivity estimates significantly more accurate than traditional methods [25, 17, 46, 47]. However, their construction overhead [46, 22] can be huge compared to the traditional statistics collection methods [41, 3] when the labeled training examples are generated synthetically. Considering that the set of expressions that require selectivity estimate during query optimization is typically much larger than the set of expressions for which labeled examples can be collected from plans executed in the past [13, 47] - the overhead in the model construction process is an important barrier in the industrial adaption of supervised models.

To elaborate, constructing a supervised model for selectivity estimation of a given query expression has two major components: (1) generating training examples with their selectivity labels, and (2) using the labeled examples to train a supervised model, with the former being the bottleneck [17, 46]. We proposed a novel method that leverages this disparity to reduce the total model construction overhead. Training data generation cost depends on two factors (1) number of training examples, and (2) per example label generation cost. In this section, we present ideas that can help significantly reduce overhead due to both of these factors.

4.1 Deciding appropriate size of training data

In general, there are no easy ways to decide the required number of training examples [38, 23]. While larger number of examples is better for prediction accuracy, it increases model construction cost. On the other hand, training with too few examples can compromise model accuracy. For a given query expression, let s^* denote the smallest number of examples such that the trained model reaches a target prediction accuracy, e.g., 95^{th} percentile q-error is below 10. We propose to use an iterative approach to increase the number of training examples, until we determine sufficient number of training examples s to achieve the target accuracy. Observe that while an iterative procedure has the potential to save the cost of generating unnecessarily more examples, it brings an additional repetitive overhead of monitoring the accuracy at the end of each iteration.

While iterative training procedures have been proposed in the past [38, 23], they explore the scenario where an enormous amount of labeled examples are present and the dominant overhead is training the model using all available examples. They use a large test set for monitoring the improvement in model accuracy and show that increasing the number of examples in doubling fashion minimizes the total time spent on model training steps. A straightforward adaption of their approach would not have worked since, in our context, label generation is the dominant overhead (also it varies across query expressions), and training overhead varies across various model designs [25, 17, 47].

4.1.1 Proposed Algorithm: IterTrain

Our proposal for iterative model construction, termed as *IterTrain*, is formalized in Algorithm 1 and visualized in Figure 1. It uses cross-validation for estimating model accuracy and computing confidence interval, and we show that the optimal geometric step size depends on the ratio between label generation cost and cross-validation cost [4].

[4]Observe that, a linear schedule would still be worse than geometric since in the linear schedule the total cost of k-fold cross-validation steps (up to any stage) grows quadratically in the number of examples and becomes dominant as the number of iterations increase.

Algorithm 1 *IterTrain*

Inputs: maximum # training examples m, target q-error $\epsilon > 1$, target percentile p, significance level of confidence $\delta < 1$, initial training size s_0, ratio r of labeling cost vs. cross validation cost per example

1: **Initialization:** Label $s = s_0$ random examples as the initial pool D. Set $c = 1 + \sqrt{\frac{1}{r+1}}$.
2: **while** $s \leq m$ **do**
3: Run cross validation using D
4: Compute the confidence interval $[\underline{p}, \overline{p}]$ of the percentile of predictions with q-error below ϵ
5: **if** $\underline{p} \geq p$ **then** break
6: $s_0 \leftarrow s$
7: **if** $\overline{p} \leq p$ **then**
8: $s \leftarrow m$
9: **else**
10: $s \leftarrow c \times s$
11: label $s - s_0$ random examples and add them to D
12: return model trained with D

To elaborate, in each iteration, we run k-fold cross validation on the training examples and use it to compute the confidence interval ($\overline{p}$ and $\underline{p}$) for the percentage of examples below q-error threshold ϵ, as explained in Section 4.1.2. When the target percentile p is equal or below the lower bound $\underline{p}$ (line 5), we know that the target has been met with probability at least $1 - \delta$. So we stop generating more training examples. When the target percentile p is equal or above the upper bound $\overline{p}$ (line 7), we know that the target cannot be met with the given budget m with probability at least $1 - \delta$, so we use the maximal budget m and return a best model we can train with this budget. When the target percentile is contained in the confidence interval (line 9-10), our algorithm increases the sample size geometrically using the optimal geometric ratio derived in Section 4.1.3.

4.1.2 Confidence intervals for accuracy target

Consider k-fold cross validation on s training examples. For each fold, we use $s \cdot \frac{k-1}{k}$ examples for training and $s \cdot \frac{1}{k}$ examples for testing. Each fold i produces an empirical estimation of the true percentage p of examples with q-error below ϵ in the test distribution. We follow the proof of Theorem 1 and 2 in Huang et al. [23], and replace the classification accuracy in [23] with the percentage p, to derive a δ-confidence interval for p as:

$$p \in \left[p_{i,2} - \sqrt{\frac{k}{2s} \ln \frac{1}{\delta}}, p_{i,1} + \sqrt{\frac{k}{2(k-1)s} \ln \frac{2}{\delta}} + \sqrt{\frac{k}{2s} \ln \frac{2}{\delta}} \right] \tag{1}$$

where $p_{i,1}$ and $p_{i,2}$ correspond to the empirical percentage of examples with q-error below ϵ in the test data and training data of fold i respectively. Given that $p - p_{i,j}, i \in [k]$ have identical distributions for $j = 1, 2$ respectively, we know that the length of the confidence interval for the mean of $p - p_{i,j}$ is $\frac{1}{\sqrt{k}}$ of that of $p - p_{i,j}$. So:

$$\left[p_2 - \sqrt{\frac{1}{2s} \ln \frac{1}{\delta}}, p_1 + \sqrt{\frac{1}{2(k-1)s} \ln \frac{2}{\delta}} + \sqrt{\frac{1}{2s} \ln \frac{2}{\delta}} \right] \tag{2}$$

is a δ-confidence interval for p.

Note that these confidence intervals are valid because we label examples uniformly at random. They do not hold if we use active learning [40] to label non-uniformly random examples.

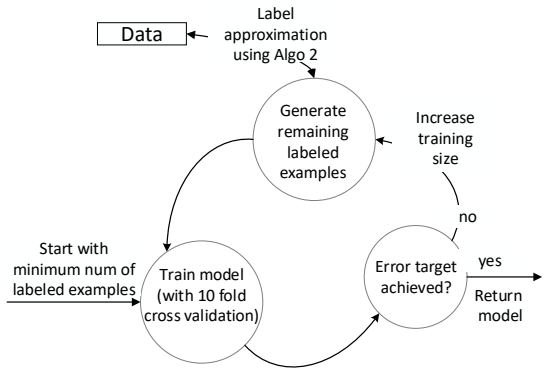


Figure 1: Visual representation of IterTrain

4.1.3 Optimizing the geometric step size

The approximate optimality of geometric scheduling was first proved by Provost et al. [38], and the optimal base value was presented by Huang et al. [23]. Our context is different since the cost of labeling dominates the cost of cross-validation, for a given number of examples. Still, we can prove that our algorithm has a constant-ratio approximation guarantee of optimality, and our choice of the geometric base c is optimal in terms of the approximation ratio.

THEOREM 1. *Assuming a specific random order of training examples, IterTrain has a* $\left(\sqrt{\frac{1}{r+1}} + 1\right)^2$*-approx guarantee, i.e., the total cost is no larger than* $\left(\sqrt{\frac{1}{r+1}} + 1\right)^2$ *times the cost of directly generating and cross-validating with* s^* *examples, where* s^* *is the optimal number of training examples such that* $s^* > s_0$.

PROOF. Let the cost of generating s training examples and cross-validation using s examples be $C_1(s)$ and $C_2(s)$ respectively. In our problem, $C_1(s) = C_2(rs) \gg C_2(s)$, and both are proportional to s. The cost of generating and cross-validating with the optimal number of s^* training examples is $C_{opt} = C_1(s^*) + C_2(s^*) = (r+1)C_2(s^*)$.

The training size used in *IterTrain* is $s_0, cs_0, \ldots$. Let z be the smallest integer such that $c^z s_0 \geq s^*$. By definition, $c^{z-1} s_0 < s^*$. The cost of *IterTrain* is given by:

$$\begin{aligned} C =& C_1(c^z s_0) + \sum_{i=0}^{z} C_2(c^i s_0) \\ =& C_1(c \times c^{z-1} s_0) + C_2\left(\frac{c^{z+1} - 1}{c - 1} s_0\right) \\ <& C_1(cs^*) + C_2\left(\frac{c^2}{c-1} s^*\right) = \frac{rc + \frac{c^2}{c-1}}{r+1} C_{opt} \end{aligned} \tag{3}$$

This means that for any geometric base $c > 1$, the algorithm has a $\frac{rc + \frac{c^2}{c-1}}{r+1}$-approx guarantee. This number reaches its minimum $\left(\sqrt{\frac{1}{r+1}} + 1\right)^2$ when $c = 1 + \sqrt{\frac{1}{r+1}}$. □

Intuitively, the theorem implies that the iterative procedure can get closer to the optimal cost of model construction

by reducing the geometric step size, compared to doubling approach [38, 23], when cross validation is significantly faster compared to label generation cost, e.g., $r \approx 100$ leads to approximation ratio < 1.2.

Finally, observe that the above theorem gives approx-guarantee with respect to C_{opt} that includes cost of k-fold cross-validation. The approx-guarantee with respect to only label generation cost is given by $c + \frac{c^2}{r(c-1)}$, which can be much higher than $\frac{rc+\frac{c^2}{c-1}}{r+1}$ when $r < 1$. This implies that, for techniques that are slower-to-train to cause r to be much less than 1, *IterTrain* with cross-validation can have large constant in the approx-guarantee and it may be better for *IterTrain* to use a hold-out test set for monitoring model accuracy across iterations (and hence increase 'r').

4.2 Efficient selectivity label approximation

As mentioned above, generating the selectivity label for a given query (line 11 in Algorithm 1) can be quite expensive for large datasets. Our proposal to reduce the cost of computing selectivity labels is based on the assumption that we can tolerate small errors in the model predictions (e.g., q-error < 2) - our experiments empirically validate this assumption. We exploit this by generating approximate selectivity labels for training queries using a data sample, rather than the full data. In general, the larger the selectivity of a query is, the smaller is the size of the data sample required to produce its approximate label, for a given relative error bound. We note here that similar ideas for selectivity approximation are used in approximate query processing (AQP) systems like BlinkDB [6], the main difference is we target to bound q-error instead of absolute error and we need to minimize total cost of label generation for multiple queries in each training iteration. With these observations, we formally describe our proposal for efficient generation of approximate selectivity labels.

Let $Q_{rem} = \{q_1, \ldots, q_n\}$ be the working set of training queries that remained unlabeled after the previous iterations of Algorithm 2, and $act(q_i)$ be the true selectivity of q_i. Using t random tuples from the table, we can obtain an approximate selectivity label $l_i(t)$ for each query q_i, i.e., $l_i(t) \approx act(q_i)$ where N is the total number of tuples in the table[5]. We use concentration inequalities to bound the difference between $l_i(t)$ and $act(q_i)$ with high probability. The difference decreases as t increases. We progressively increase the data sample size t used for the approximate labels. Observe that, this procedure requires the tuples to be processed in random order and scans each of the N tuples at most once. To control the labeling cost for large datasets, we may also consider an upper limit Upper_Limit on the number of tuples to be processed. When the difference for a query q_i is below a desired threshold with high probability, we drop the query q_i from the working set of queries and stop updating its approximate label. When the working set of queries is empty, we obtain the approximate labels for all training queries. The procedure for approximate label generation is formalized in Algorithm 2.

Given query q_i and sample size t, we now describe how to calculate the confidence probability (line 4) for $l_i(t) \in [\frac{act(q_i)}{\eta}, \eta \times act(q_i)]$. To simplify notations, we use l and a to abbreviate for $l_i(t)$ and $act(q_i)$.

[5]Observe that, t denotes the count of all tuples processed until the current iteration (equivalent to the value of y in line 3 of Algorithm 2).

Algorithm 2 *ApproxLabel*

Inputs: $Q_{rem} = \{q_1, \ldots, q_n\}, T = \{t_1, \ldots, t_N\}$, q-error threshold $\eta > 1$, probability threshold $\alpha < 1$
1: **Initialization:** $x = 0, y = 100$, tuples in T are assumed to be in random order
2: **while** $y \leq \min(\text{Upper\_Limit}, N)$ and $Q_{rem} \neq \emptyset$ **do**
3: Evaluate tuple t_{x+1} to t_y for each query q_i in Q_{rem} and update their approximate labels
4: Update confidence probability for each query q_i in Q_{rem} to achieve q-error bound η
5: Drop queries from Q_{rem} for which the confidence probability is above α
6: $x \leftarrow y$
7: $y \leftarrow 2 \times y$

First, based on the multiplicative Chernoff bound, we have:

$$Pr\left[\frac{l}{N} < \frac{a}{\eta N}\right] < \left(\eta^{1/\eta} e^{1/\eta - 1}\right)^{ta/N} = \left(\eta e^{1-\eta}\right)^{\frac{ta}{\eta N}} \quad (4)$$

$$\leq \left(\eta e^{1-\eta}\right)^{tl/N} \quad (5)$$

The last inequality is because $\eta > 1$, $\eta e^{1-\eta} < 1$.

Second, based on Hoeffding's inequality, we have:

$$Pr\left[\frac{l}{N} - \frac{a}{N} > \frac{l}{N} - \frac{l}{N\eta}\right] \leq e^{-2t^2 l^2 (1-1/\eta)^2 / N^2} \quad (6)$$

Together, with probability at least $1 - \left(\eta e^{1-\eta}\right)^{tl/N} - e^{-2t^2 l^2 (1-1/\eta)^2/N^2}$, the ratio between approximate label l and the true selectivity a is bounded by $[1/\eta, \eta]$.

It is easy to see that the total number of samples used by *ApproxLabel* has a constant ratio approximation guarantee of optimality.

PROPOSITION 1. *ApproxLabel has a 2-approx guarantee, i.e., the total labeling cost is no larger than twice the cost of using the minimal number of samples for each query.*

Observe that, while sample size in Algorithm 2 follows a geometric schedule, a linear schedule would have been a better choice when N is very large and most of the queries happen to have large values of actual selectivities, which implies that small data samples would be sufficient.

4.3 Practical considerations

Observe that, Algorithm 1 can ensure that target accuracy is achieved if (a) maximum training size m is sufficiently large, (b) the model being used has sufficient representative power, and (c) true selectivity labels are used for training. But due to practical constraints on training time or memory footprint, one or more of these requirements may not satisfy. In such cases, the training procedure can be restarted with larger values of m or model size, while reusing the labeled examples already generated in the first attempt. Further, if we choose to use small value of Upper_Limit on sample size, it can lead to errors in selectivity labels. In that case, the learned model is approximating the selectivity function for a random data-sample with Upper_Limit rows.

Algorithm 1 can use any regression technique for model construction. Use of such alternatives would only cause change in value of parameter r (ratio of labeling cost to cross validation cost), resulting in different optimal value of

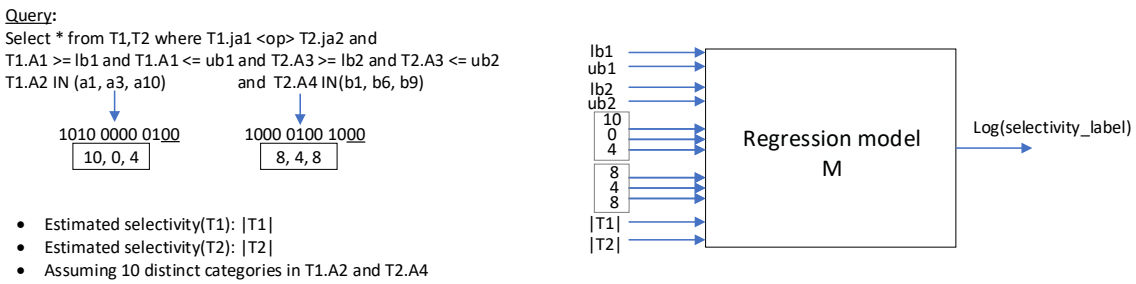


Figure 2: Model design for an example query

geometric step size c. Algorithm 2 is independent of the method used to compute selectivity labels (line 3 in Algorithm 2). We use regular query execution to get these labels, but it can easily be replaced by a more efficient alternative - for instance, an unsupervised model [48].

An important issue with these models is that of incremental maintenance when there are updates to underlying data. Such update invalidate the selectivity labels used for training of supervised models. This is an interesting problem beyond the scope of this paper. Observe that, our efficient training method that help reduce model construction cost makes the update issue less severe.

Finally, observe that with increase in iterations, the labeling cost keeps increasing while the accuracy gains may diminish. It may be possible to use the intermediate learned model to compute approximate labels for a subset of new training examples to further reduce labeling costs - we defer such attempts for future work.

5. MODEL DESIGN

We present our model design choices for a given query expression. We consider join expressions between two or more base tables, with simple filter ($column\ \langle op \rangle\ constant$). Specifically, we consider equality, range and IN filter types and the join predicates may vary from key-foreign key, equality or even inequality types. Our model design choices are driven by small training time and fast estimation time.

5.1 Basic design

We inherit the following basic aspects from [17]. We use XGBoost [14] as the regression technique with *log-transformed* selectivity labels. The reason for these choices are (1) log-transformed labels help in better accuracy when the evaluation metric is relative (q-error) and, (2) training time for tree-based ensembles is much faster compared to multi-layer perceptron regressor for a given training set [17].

The evaluation in [17] was limited to multiple range predicates on a single base table. By treating the relation corresponding to a query expression as a table itself, we can use the same basic design for intermediate expressions. Figure 2 illustrate our adaption of regression models in [17] for a join between two tables with range as well as IN filter on each of the base tables.

We now describe our input feature encoding to regression model M given a query example, also shown visually in Figure 2. Observe that, the set of base tables participating in the join and the join-predicates need not be used as explicit input features since all the training examples correspond to the same join expression. Next, we discuss our input features choices for filter predicates.

5.2 Query features

Range predicates For each range filter specified by lb_i and ub_i, we use the values lb_i and ub_i to featurize such range predicate. This encoding is also inherited from [17]. The fact that the range filters may appear on different side of the join $T1 \bowtie T2$ has no impact on its representation.

Categorical IN predicates For filter predicates of the form $A2$ IN ($catg\_list_1$), we first compute a bitmap that encodes the domain values that appear in the list ($catg\_list_1$). Further, we perform a decimal encoding of the bitmap, by first splitting the bitmap into equal sized chunks and then convert each chunk of bits into its decimal form. We use the resulting sequence of decimal numbers as input feature to the model. For example, as shown in Figure 2, categorical attribute $A2$ with 10 values in its domain a_1 through a_{10} has an IN clause (a_1, a_3, a_{10}). For this filter, the bitmap is given by 101000000100. Note that, last two bits (underlined) have been added to make the bitmap length a multiple of chunk size[6]. The decimal encoding (10,0,4) become three input features to the model. We use decimal encoding to reduce the number of input features to the model. Similarly, IN filter on $A4$ leads to three more input features as shown in Figure 2. If the number of categories in an attribute domain is very large, it would help to exploit distribution skew and keep most frequent categories in the bitmap, and hash the infrequent categories into smaller number of dimensions [33]. Note that, computation of both range and categorical features (for moderate size domains) is efficient and requires information from either the query or pre-computed statistical metadata.

5.3 Selectivity features

While query features are sufficiently descriptive, it helps to add additional relevant features – specifically, when we want to use small sized models to keep estimation time low [17, 25]. We add per-table selectivity estimates as extra inputs to the model. These features are computed from compile-time statistics, such as histograms, that are typically available in most database systems. These estimates can certainly be replaced by more accurate estimates from regression models, if available, for the participating base tables. Our selectivity features are in the same spirit as

[6]In our experiments we use chunk size 32, details in Section 6.5.

CE features [17] or sample-based estimates [25], and are expected to help improve accuracy while adding small overhead to the estimation time. Overall, our model for example query q has 12 features including four range features, six categorical features and two selectivity features, as shown in Figure 2.

5.4 Discussion

Observe that the above discussion is not limited to the case for a single join $T1 \bowtie T2$. A similar method can be followed to design a model for join expressions with more than two tables, by featurizing the range and categorical predicates across all the participating tables and adding a selectivity feature for each table. We consider different combinations of features (1) query features only; (2) selectivity features only; and (3) query and selectivity features together, to find the best set of features across many different query expressions.

Sample-bitmap features. Our empirical analysis of the benchmark (TPC-H, TPC-DS) and industrial workload show that the filter types natively supported by our model design (equality, range, categorical) are sufficient to cover a large majority of the queries. To support more complex filter predicates, we include evaluation of the per-table sample bitmap features proposed in [25]. Empirically, our observation is same as [25] that sample-bitmaps can help in improving higher percentile errors even when queries have simple filters. Given their added estimation overhead, we evaluate a feature sub-selection approach to extract their accuracy benefits with much smaller estimation overhead. To elaborate, we first train the model using a large number of bits (say 10,000) in the sample-bitmap, and then rank these bits in terms of feature importance. While preparing the final model we use only the high rank bits, and ignore the rest to bring down the estimation overhead.

5.5 One model for multiple expressions

Until now, we have described model design for the simplest case of a single join expression. Observe that, the model works with a single relation that can correspond to any join expression. It also means that in the scenarios where a materialized view of a join expression can support queries for different subsets of tables, we can also learn a single model to estimate selectivities for different join subexpressions. For instance, consider the case when a fact table T_F joins with two dimension tables T_{D1} and T_{D2} and we wish to estimate selectivities for all 3 join sub-expressions $T_F \bowtie T_{D1}$, $T_F \bowtie T_{D2}$, $T_F \bowtie T_{D1} \bowtie T_{D2}$, with pre-specified template filters on T_F, T_{D1}, T_{D2}, etc. In this case, it is possible to learn a single model that can serve the purpose[7]. Specifically, we can estimate selectivity of $T_F \bowtie T_{D1}$, by setting the filter predicates on T_{D2} as full-domain predicates. In Section 6, we present an experiment where we use a single 16KB model to estimate selectivities of 15 join subexpressions extending from 2-way to 5-way joins. Observe that, this model can also generate single table selectivity estimates for the fact table T_F, but not for the dimension tables T_{D1} and T_{D2}.

[7]Similar to how a join-sample constructed using a sample from the fact table can serve estimates for many join subexpressions [5]

6. EXPERIMENTAL EVALUATION

We evaluate the proposed models on construction overhead, estimation accuracy, feature set effectiveness, as well as impact of injecting model estimates on quality of plans chosen by the optimizer.

6.1 Experimental setup

We include the following techniques in our evaluation,

1. ComOpt, a commercial optimizer that uses per-attribute histograms on base tables and black-box complex computations to compute selectivity estimates for join-expressions.
2. *Statistics on views [8, 19] (abbv. as* SOV*)*, use per-attribute histograms on the view for the join expression.
3. *Join-sample (*JS(10k)*)*, a 10 thousand row uniform random sample over the result of the join expression. We also use 100 thousand row join-sample (abbv. as JS(100k)) as accuracy baseline.
4. MSCN, a recently proposed [25] neural network based supervised learning model design. We use code sourced from authors [1] with recommended values for hyper-parameters (100 epochs, 256 neurons).
5. *Regression model variants (model_X)*, the models proposed in this work with feature choices captured in 'X', where 'Q' represents query features, 'E' represents selectivity features and 'B' represents sample-bitmap features.

Setup and estimation overhead. For all techniques that require materialized join expressions, we use a 100 thousand row join sample constructed over the result of the join expression. Effectively, learned models are approximating the selectivity function from this 100 thousand row sample.

We use 10 thousand training examples per join expression to train MSCN models, as well as an upper limit on training size for the proposed models. We report training time and accuracy of training a MSCN model for each individual join expression to achieve best model accuracy, sometimes better than a global model training across all expressions [25]. The sample-bitmap features for MSCN or model_QBE are computed using 1000 row uniform random sample over each base table, similar to [25]. XGBoost is used with following hyper-parameters: {*tree_method:hist*, *grow_policy:lossguide* }.

In terms of estimation overhead, ComOpt as well as SOV require only tens of KBs and $< 100\mu sec$ for scanning the per-attribute histogram structures. The proposed XGBoost model (model_QE) uses only 16 KB memory (16 trees with max. 16 leaves each) with estimation time $\approx 100\mu sec$ per inference (similar to [17]). As expected, models that use base table sample bitmap features need extra time for sample bitmap computation. And the estimation time for JS(10k) is *a few milliseconds.*

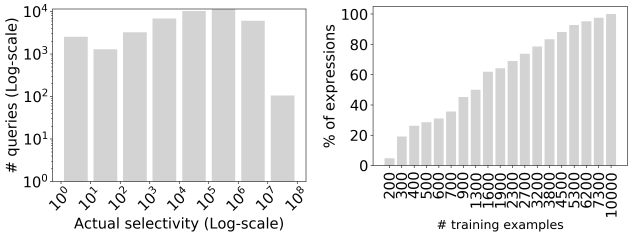
Queries and datasets. We use 42 join expressions across 3 datasets in our evaluation, with a number of tables in the join - later referred to as *join-dimension* - varying from 2 to 5. For each join-expression, we generate query instances by adding a range predicate and an IN predicate on each of the participating tables. Thus, our queries have a predicate count varying from 4 to 10. While real-world queries can have more tables, having multiple filters on each base table makes our setup sufficiently challenging for join selectivity estimation task.

To create a query with non-empty result, we use a randomly chosen row from the 100 thousand row data sample as the seed - we call it a *data-centric* query. Given a data row seed, different queries may differ in their width for range predicates and number of categories added to the IN lists. The join-expressions are chosen from the following datasets: TPC-H with skew (1GB, z=2), TPC-DS (10GB) and a real sales dataset (97 GB).

For the purpose of accuracy evaluation, we generated 1000 queries for each expression using the same query generation parameters as training examples. Note that, the actual selectivities of test queries are computed over the entire materialized join (not the 100 thousand row sample used for the estimators). Figure 3(a) shows the distribution of actual selectivities for test queries across all expressions. Observe that the actual selectivities are spread over large range of values to avoid bias against any technique. Such wide distribution of selectivities also presents a variety of learning challenge to the models.

6.2 Effectiveness of iterative procedure

As mentioned above, we use an upper limit of 10 thousand examples for each join expression, and initial training size of 100 examples. We aim to achieve a target accuracy of $p = 95$ percent queries below q-error of $\epsilon = 10$. We use geometric step size 1.2 in our experiments, and increase training size only in multiples of 100.

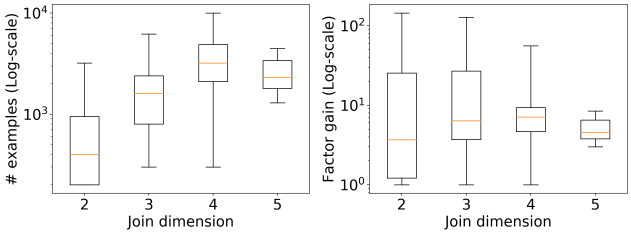


(a) Actual Selectivities (b) # training examples

Figure 3: (a) Distribution of actual selectivities across all expressions (b) Number of training examples used by IterTrain vary significantly across expressions

To evaluate the effectiveness of iterative procedure, we evaluate the number of training examples used out of the 10 thousand budget, across different join expressions. Figure 3(b) shows cumulative percentage of join expressions with increasing training size. Observe that for around 40% of the join expressions, less than 1000 examples were found sufficient to meet the target accuracy and 4000 examples were sufficient for 75% of the expressions. There are expressions for which the selectivity function was "difficult" to learn and they required 5000 or more examples for training. Note that we use additional practical exit criteria in addition to that specified in Algorithm 1, that is, when $\underline{p}$ does not improve in last 5 iterations. Overall, we can conclude that using 10 thousand or more training examples, would be overkill for most of the expressions. On the other hand, if we used 1 thousand examples for all expressions, then we could loose on model accuracy for many expressions.

In Figure 4(a), we group together join expressions according to their join-dimension and observed that there is a consistent increase in the number of training examples with the



(a) # training examples (b) Factor accuracy gain

Figure 4: Effectiveness of IterTrain w.r.t. dimensions

increase in join-dimension. In our experimental setup, one reason for this is that the number of filter predicates also increase with the number of joins. However, the training size required for a fixed dimension also varied significantly. It implies that dimension can not be used as a direct proxy to decide the size of training data.

To evaluate the improvement in accuracy, we use the ratio of q-error values - termed as *factor-gain*. In Figure 4(b), we plot the factor gain in 95^{th} percentile values from first iteration to the last iteration, grouped by join dimension. It shows that for many expressions, the 95^{th} errors improve significantly, i.e., by a factor of 10-100×.

6.3 Model construction overhead

In Figure 5, we illustrate how worst case *IterTrain* overhead stand compared to one-shot training with 10k examples (using MSCN with Integrated GPU0 Quadro K420) and the lower bound of generating only necessary training examples. Further, we demonstrate different *IterTrain* scenarios depending on whether: (i) 10-fold cross-validation or hold-out testing (with 2k examples) is used for monitoring accuracy; (ii) the model training set is relatively fast (XGBoost) or slow (MSCN); and (iii) per-label cost is relatively large (10 sec) or small (0.1 sec). We find that when per-label cost is relatively high, *IterTrain* with cross-validation delivers small constant-factor approximation to lower bound for both MSCN and XGBoost. However, when the per-label cost is smaller, e.g., due to approximation, it is better for MSCN to use hold-out testing since 10-fold cross-validation increases the per-example training cost. On the other hand, overhead with XGBoost remains close to the lower bound due to its significantly faster training routine.

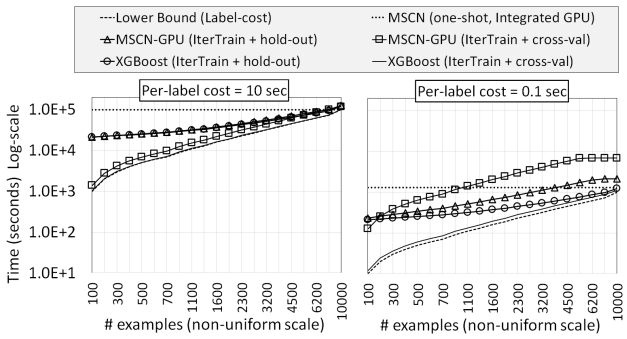


Figure 5: Overhead analysis for IterTrain scenarios

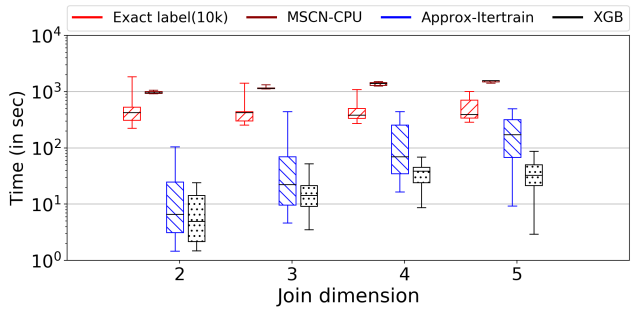


Figure 6: Construction overhead comparison with one-shot training w.r.t. join-dimension (on CPU)

We now report the cost spent in model construction when CPU is used for training both models, in Figure 6. For MSCN, we report the time spent in computing selectivity labels from the 100 thousand row join sample for 10 thousand queries (shown as *Exact label (10k)*) and then time spent in training MSCN (shown as MSCN-CPU) (GPU needs 7×-13× lesser time). For the proposed models, we report the time taken by our Algorithm 1 while approximating labels using Algorithm 2 - this is represented as *Approx-Itertrain*. Finally, we also report the total time spent in cross-validation (and model training) with XGBoost across all iterations. Overall, the total construction cost for our models typically varied from tens of seconds to a few hundred seconds, with the worst case being less than 1000 seconds. In contrast, one-shot MSCN model training with 10 thousand examples need more than 1000 seconds and label generation adds another several hundred seconds. Note that use of 100k join sample itself reduces the overhead of label generation by huge factor. The additional savings due to *Approx-Itertrain* vary from 2-20×, with an order of magnitude saving for a large fraction of expressions. Observe that the time spent in XGBoost training never exceeded a few minutes across multiple iterations with ten-fold cross validation in each iteration. For majority of expressions, XGBoost took only tens of seconds.

Note that MSCN training overhead can be reduced by tuning the model design, e.g., decreasing neuron count of feature count or number of epochs, but that may hurt the model accuracy. Parallel resources also can be leveraged to reduce the latency of labeling phase, independent of the training method. Overall, XGBoost models can achieve similar accuracy with much smaller total resource consumption.

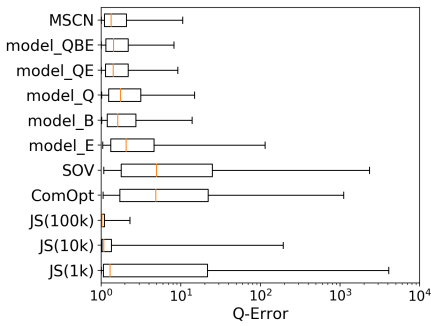
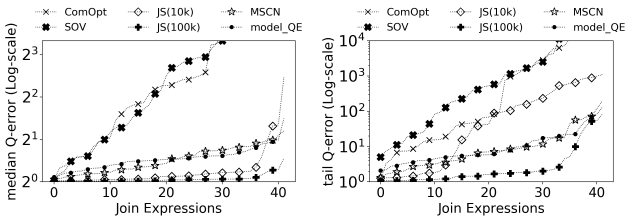


Figure 7: Estimation error comparison across techniques



(a) Median Q-error (b) 95^{th} Q-error

Figure 8: Expression-wise accuracy comparison

6.4 Estimation error evaluation

Next, we evaluate the estimation accuracy of the proposed regression models against existing techniques. Figure 7 shows error-distribution for each technique across all expressions used in our evaluation. We plot only the errors in the up to 95 percentile to avoid the outlier cases. The first highlight of this plot is that all learned models deliver more accurate estimates when compared to ComOpt, SOV and 1000 row join sample. These are the techniques with small estimation overhead comparable to single table regression models [17]. Compared to JS(10k), learned models have better 95^{th} percentile error but worse median error.

Among learned models, model_QE provides best overall error distribution, much better than model_E and model_Q. Addition of sample-bitmap features does not lead to significant improvement across all estimates. As expected, JS(100k) provides the best accuracy and model_QE learns a reasonable approximation of its selectivity function.

Errors for different expressions. Next, we report per-expression median and 95^{th} percentile q-error in Figure 8(a) and Figure 8(b) in log-scale q-error. Figure 8(a) shows the median q-error plot for each technique sorted across different expressions. We find that both ComOpt or SOV are worst in terms of median error, all other techniques are significantly better, with JS(10k) and JS(100k) being the best. Both learned techniques MSCN and model_QE are typically within 2× of JS(10k) in terms of median error.

Similarly we plot 95^{th} percentile q-errors in Figure 8(b). We find that ComOpt and SOV have 95^{th} q-error larger than 100 for many expressions, and SOV does not always improve upon ComOpt. Even JS(10k) can have large values of 95^{th} percentile q-error. All learned techniques meet the accuracy target for most of the expressions. The target could not be achieved in the remaining cases due to errors in approximated labels, as evident by the spike in JS(100k) curve.

We highlight here that the relatively worse estimation accuracy of MSCN in a few cases does not imply that their model design is not capable of good estimation accuracy. Following are the possible reasons for the large 95^{th} percentile errors: (a) IN predicates need to be explicitly featurized in their design as well, (b) MSCN is more sensitive to approximated labels or needs hyperparameter optimization.

Accuracy improvement due to proposed models. Figure 9 summarizes the accuracy improvement achieved by model_QE compared to each of the other 4 techniques in the form of *factor gain* in 95^{th} percentile q-error for individual

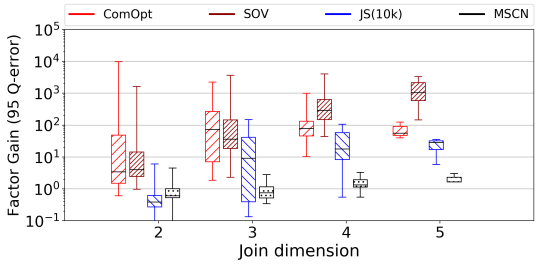


Figure 9: Factor gain in 95th percentile for proposed 16KB XGBoost model compared to other techniques

expressions. We plot the factor gain with increase in the number tables in the join, with observations as follows,

1. For majority of cases, model_QE is 10× to 100× more accurate compared to ComOpt and SOV.
2. JS(10k) can be better than model_QE (factor < 1) in some cases, but model_QE is 10× or more accurate in majority of the cases. For 2-table joins, JS(10k) is particularly better than model_QE. The reason for such specific behavior is our experiment setup that has at most 2 filter predicates per table, leading to larger values of selectivities for queries over 2-table joins compared to joins with more tables.
3. The factor gain increases consistently with the join dimension, this is because the estimation errors for traditional techniques get worse with more joins (except for MSCN).
4. We do not observe significant regressions in 95^{th} accuracy compared to MSCN, even though model_QE is significantly smaller (16 KB) compared to MSCN (3 MB). It is possible that MSCN could be more accurate with explicit IN features and hyper-parameter tuning.

Overall, we found that model_QE with only 16 KB size produces accurate estimates, orders of magnitude better than non-ML approaches and comparable to more complex MSCN model.

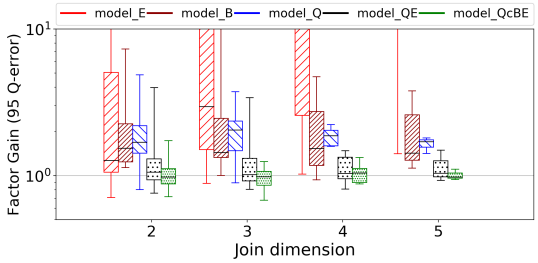


Figure 10: Factor gain in 95th percentile for proposed 16KB XGBoost model compared to feature variations

6.5 Impact of features choices

In Figure 10, we report factor gain in 95^{th} percentile q-error for a model variant with all features (model_QBE) compared to different variants with increasingly more features. We found that that using all features is significantly better than any variant with only one kind of features (model_E, model_B, model_Q). In fact, combining only compile-time features (model_QE) is significantly more accurate compared to using only sample-bitmap features (model_B) that indirectly encodes both query and selectivity features, highlighting that it is better to add features explicitly whenever possible. Finally, we found that a variant that uses only 100 highest rank bits from the sample-bitmap in terms of feature importance (model_QcBE), brings most of the accuracy gains of full-sample bitmap (1000 bits per base table), at significantly reduced run-time overhead. Overall, model_QE is a reasonable model choice without any run-time feature, while model_QcBE provides good balance between the accuracy and the estimation overhead and also support any complex filter predicate.

Impact of decimal encoding for IN features We now present an experiment where we evaluate IN clause features with different chunk-sizes for decimal encoding, including no decimal encoding (chunk-size=1) as well as absence of the IN features. Figure 11 summarizes the factor gain in tail-accuracy achieved by using IN features without any decimal encoding as compared to others variants. Largest gains ($> 10\times$) compared to model_RE shows that using only range features (no IN features) significantly degrades model accuracy. Large gains ($\approx 10\times$) compared to model_RB shows that including sample-bitmap features (as an alternative to explicit IN features) is not enough. Finally, rare and small gains compared to model_QE (chunk-size 32) and model_Q8E (chunk-size 8) show that our choice of chunk size works reasonably well in most cases. Notwithstanding, using a relatively small chunk-size, say 8, may be a safer approach.

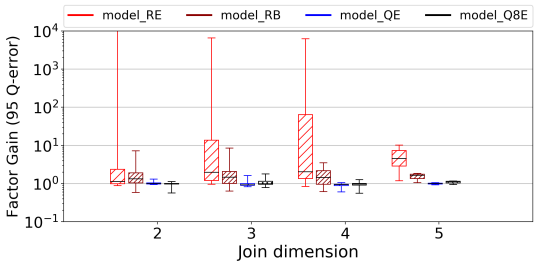
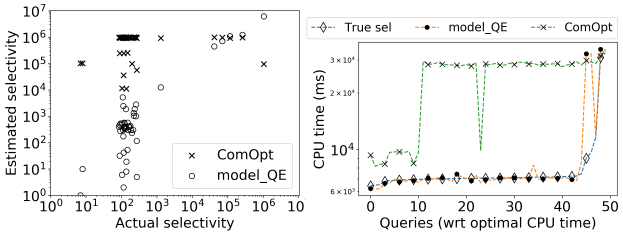


Figure 11: Impact of IN features and their decimal encoding

6.6 Plan quality experiment

Finally, we evaluate the impact of using estimates generated using model_QE, compared to using true selectivities during query optimization on the quality of chosen plans in terms of CPU time improvement w.r.t. using ComOpt estimates.

In our first experiment, we use a query based on TPC-H Q8 (joins 8 relations) and filters on *p_size*, *p_type*, *l_shipmode*, *and o_orderdate*. Due to skewed version of TPC-H dataset (z=2), we found a huge correlation across the join between *part* and *lineitem* tables. We generated 50 instances of the query with varying filters on *p_type* and *p_size*. As shown in Figure 12(a), the original estimate by ComOpt remain $\approx 10^5$ irrespective of the filter instantiations. In contrast, model_QE captures the correlation to produce orders of magnitude better estimates. When these estimates are utilized in plan selection, we get plans with 80% smaller CPU time in many cases, as shown in Figure 12(b). It is only in a few cases that estimates from model_QE do not bring the same improvement as true selectivity w.r.t. plan quality.



(a) Estimation quality (b) CPU time comparison

Figure 12: Estimation and plan quality improvement by using model_QE estimates for the expression for 50 queries based on TPC-H Q8 on skewed TPC-H (z=2)

In another experiment, we executed 500 instances of a 5 join query on the real world customer dataset, with range as well as categorical filter on each of the 5 tables. Note that during optimization, these queries require selectivity estimates for various 2,3,4-way joins as well. Since there was a single fact table and all joins were key-joins, we could use a *single 16 KB model* to produce all required estimates. When we injected model estimates, the plan choice changed for 74% of the queries with 30% queries resulting in a plan at least 1.2x better than the original plan choice (with ComOpt estimates). In fact, CPU time for 10% of the queries improved by 10x or more with a few queries gaining in excess of 100x. Overall, these experiments demonstrate that the significant improvements in estimation quality can lead to savings in CPU execution time as well, even when the model estimates have some errors w.r.t. true selectivity.

7. RELATED WORK

Selectivity estimation is a challenging problem for many reasons such as the expectation of small estimation and construction overhead, as well as small enough estimation errors to avoid bad plan choices. In this work, we focus on supervised models for selectivity estimation as they can support low overhead estimation with reasonably accurate selectivity estimates. We propose methods to significantly reduce the construction overhead, and demonstrate that fast-to-train models can be extended to support join-expressions with base-table filters, which is an important subclass of query expressions. Past work related to construction overhead reduction has already been discussed in Section 4 - here we focus on the alternative methods for selectivity estimation.

For query-expressions with base table filters, sampling based methods are arguably the most promising line of work in terms of estimation accuracy. These include techniques that use a join-sample for one or more join expressions [5] by leveraging key-based joins and novel techniques that efficiently produce a join-sample by sampling base-table rows [12, 49, 45, 27] etc. While sampling based techniques are great in terms of both accuracy and range of applicability, their estimation and storage overhead make them less attractive in the context of selectivity estimation for low overhead query optimization [11]. This is evident by limited adoption of such methods, e.g., [35, 36] supports use of a small uniform random sample for base tables. Actually, the proposal in this paper can be seen as an attempt to approximate the selectivity function from a large join sample and making it available at low estimation overhead for selectivity estimation. In fact, efficient techniques to construct join sample [50] can help our technique by reducing the overhead of large join-sample creation step.

Many other innovative proposals such as multi-dimensional histograms, graphical models [20, 44] or unsupervised methods [48, 22], can also provide improved estimation accuracy at the cost of large construction or estimation overhead when dealing with high-dimensional data. Interestingly, an unsupervised method [48, 22] with highly accurate estimation quality could potentially be combined with low estimation overhead of supervised methods, to get the best of both worlds.

While unsupervised methods aim to capture the entire data distribution, supervised models focus to learn selectivity function from a given set of labeled query examples, which is similar to self-tuning approaches [4, 10, 42, 21]. Once the set of queries is known, highly optimized implementations for regression techniques [14, 24] ensure that the training process is much faster compared to traditional proposals [10, 42, 21]. Recently, efficient algorithm for training uniform mixture models have also been developed [37]. We show that the faster training routine plays an important role in reducing the total model construction cost.

The idea of selectivity estimation using supervised learning is not new [26, 30, 29]. Recent work [17, 25, 46] on supervised models for selectivity estimation generate a large amount of training data to ensure good model accuracy. However, the costly step of generating sufficient training data has been a critical bottleneck in their usage [22]. In this work, we propose an iterative approach to generate training data that reduces the total model construction cost. While the proposed iterative model construction procedure can be used for any supervised model design, it is more suitable for model designs that support low overhead training routine. We also demonstrate that low overhead regression models proposed in [17] can support selectivity estimation for much broader class of query expressions.

8. CONCLUSION

Supervised models for selectivity estimation are attractive due to their low estimation overhead and ability to adapt to recent query workload. The techniques proposed in this paper improve the usability of simple regression models for selectivity estimation by making significant progress in reducing the total model construction cost and broadening the class of supported queries. Note that, the term "usability" also implies other properties such as ability to debug and explain the predictions from regression models. These properties are also important in selectivity estimation context, and leveraging the tree-based ensembles for this purpose is an area of future work. It could also be helpful to develop mechanisms that can raise an alert whenever the model generated estimate has large error.

Overall, we believe that this paper pushes the case forward for use of supervised learning methods to improve state-of-the-art in selectivity estimation.

Acknowledgments. We thank Srikanth Kandula, and the anonymous reviewers for their thoughtful comments and valuable suggestions on this work.

9. REFERENCES

[1] https://github.com/andreaskipf/learnedcardinalities.

[2] How the planner uses statistics. https://www.postgresql.org/docs/current/row-estimation-examples.html.

[3] Understanding optimizer statistics with oracle database 18c. https://www.oracle.com/technetwork/database/bi-datawarehousing/twp-stats-concepts-0218-4403739.pdf.

[4] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. In *SIGMOD*, 1999.

[5] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *SIGMOD*, 1999.

[6] 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 *Eurosys*, 2013.

[7] B. Babcock, S. Chaudhuri, and G. Das. Dynamic sample selection for approximate query processing. In *SIGMOD*, 2003.

[8] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *SIGMOD*, 2002.

[9] N. Bruno and S. Chaudhuri. Efficient creation of statistics over query expressions. In *ICDE*, 2003.

[10] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: A multidimensional workload-aware histogram. In *SIGMOD*, 2001.

[11] S. Chaudhuri. Query optimizers: Time to rethink the contract? In *SIGMOD*, 2009.

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

[13] S. Chaudhuri, V. Narasayya, and R. Ramamurthy. A pay-as-you-go framework for query execution feedback. *PVLDB*, 1(1):1141–1152, 2008.

[14] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *KDD*, 2016.

[15] Y. Chen and K. Yi. Two-level sampling for join size estimation. In *SIGMOD*, 2017.

[16] G. Cormode, M. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for massive data: Samples, histograms, wavelets, sketches. *Found. Trends databases*, 4(1-3), 2012.

[17] A. Dutt, C. Wang, A. Nazi, S. Kandula, V. R. Narasayya, and S. Chaudhuri. Selectivity estimation for range predicates using lightweight models. *PVLDB*, 12(9):1044–1057, 2019.

[18] C. Estan and J. F. Naughton. End-biased samples for join cardinality estimation. In *ICDE*, 2006.

[19] C. A. Galindo-Legaria, M. Joshi, F. Waas, and M. Wu. Statistics on views. In *VLDB*, 2003.

[20] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *SIGMOD*, 2001.

[21] M. Heimel, M. Kiefer, and V. Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *SIGMOD*, 2015.

[22] B. Hilprecht, A. Schmidt, M. Kulessa, A. Molina, K. Kersting, and C. Binnig. Deepdb: Learn from data, not from queries! *PVLDB*, 13(7):992–1005, 2020.

[23] S. Huang, C. Wang, B. Ding, and S. Chaudhuri. Efficient identification of approximate best configuration of training in large datasets. In *AAAI*, 2019.

[24] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T.-Y. Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *NIPS*, 2017.

[25] A. Kipf, T. Kipf, B. Radke, V. Leis, P. A. Boncz, and A. Kemper. Learned cardinalities: Estimating correlated joins with deep learning. In *CIDR*, 2019.

[26] M. S. Lakshmi and S. Zhou. Selectivity estimation in extensible databases - a neural network approach. In *VLDB*, 1998.

[27] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality estimation done right: Index-based join sampling. In *CIDR*, 2017.

[28] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. Boncz, A. Kemper, and T. Neumann. Query optimization through the looking glass, and what we found running the join order benchmark. *The VLDB Journal*, 27(5), 2018.

[29] H. Liu, M. Xu, Z. Yu, V. Corvinelli, and C. Zuzarte. Cardinality Estimation Using Neural Networks. In *CASCON*, 2015.

[30] T. Malik, R. Burns, and N. Chawla. A black-box approach to query cardinality estimation. In *CIDR*, 2007.

[31] G. Moerkotte, D. DeHaan, N. May, A. Nica, and A. Boehm. Exploiting ordered dictionaries to efficiently construct histograms with q-error guarantees in sap hana. In *SIGMOD*, 2014.

[32] G. Moerkotte, T. Neumann, and G. Steidl. Preventing bad plans by bounding the impact of cardinality estimation errors. *PVLDB*, 2(1):982–993, 2009.

[33] J. Moody. Fast learning in multi-resolution hierarchies. In *NIPS*, 1989.

[34] M. Müller, G. Moerkotte, and O. Kolb. Improved selectivity estimation by combining knowledge from sampling and synopses. *PVLDB*, 11(9):1016–1028, 2018.

[35] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*, 2020.

[36] Optimizer statistics (release 18). https://docs.oracle.com/en/database/oracle/oracle-database/18/tgsql/optimizer-statistics.html#GUID-0A2F3D52-A135-43E1-9CAB-55BFE068A297.

[37] Y. Park, S. Zhong, and B. Mozafari. Quicksel: Quick selectivity learning with mixture models. In *SIGMOD*, 2020.

[38] F. Provost, D. Jensen, and T. Oates. Efficient progressive sampling. In *SIGKDD*, 1999.

[39] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, 1979.

[40] B. Settles. Active learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 2009.

[41] Statistics in Microsoft SQL Server 2017. https://docs.microsoft.com/en-us/sql/relational-databases/statistics/statistics?view=sql-server-2017.

[42] U. Srivastava, P. J. Haas, V. Markl, M. Kutsch, and T. M. Tran. Isomer: Consistent histogram construction using query feedback. In *ICDE*, 2006.

[43] M. Stillger, G. Lohman, V. Markl, and M. Kandil. Db2's learning optimizer. In *VLDB*, 2001.

[44] K. Tzoumas, A. Deshpande, and C. S. Jensen. Efficiently adapting graphical models for selectivity estimation. *The VLDB Journal*, 22(1), 2013.

[45] D. Vengerov, A. C. Menck, M. Zait, and S. P. Chakkappen. Join size estimation subject to filter conditions. *PVLDB*, 8(12):1530–1541, 2015.

[46] L. Woltmann, C. Hartmann, M. Thiele, D. Habich, and W. Lehner. Cardinality estimation with local deep learning models. In *aiDM@SIGMOD*, 2019.

[47] C. Wu, A. Jindal, S. Amizadeh, H. Patel, W. Le, S. Qiao, and S. Rao. Towards a learning optimizer for shared clouds. *PVLDB*, 12(3):210–222, 2018.

[48] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, P. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep unsupervised cardinality estimation. *PVLDB*, 13(3):279–292, 2019.

[49] F. Yu, W.-C. Hou, C. Luo, D. Che, and M. Zhu. Cs2: A new database synopsis for query estimation. In *SIGMOD*, 2013.

[50] Z. Zhao, F. Li, and Y. Liu. Efficient join synopsis maintenance for data warehouse. In *SIGMOD*, 2020.