Estimating Filtered Group-By Queries is Hard: Deep Learning to the Rescue

Andreas Kipf
Technical University of Munich
kipf@in.tum.de

Michael Freitag
Technical University of Munich
freitagm@in.tum.de

Dimitri Vorona
Technical University of Munich
vorona@in.tum.de

Peter Boncz
Centrum Wiskunde & Informatica
boncz@cwi.nl

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

Alfons Kemper
Technical University of Munich
kemper@in.tum.de

ABSTRACT

While estimating the result size of a group-by operation on a base table is hard on its own, the presence of selections makes this problem increasingly difficult to solve. We show that skewed data distributions and correlations found in real-world data heavily affect the results of traditional cardinality estimators. On the other hand, deep learning has recently been shown to be a more robust approach to cardinality estimation. Our evaluation shows that our (set-based) deep learning model significantly enhances the quality of filtered group-by cardinality estimates.

1. INTRODUCTION

The estimation accuracy of intermediate query result sizes dominates the plan quality of cost-based query optimizers, and bad cardinality estimates can lead to disastrous performance [24, 26]. One particularly hard problem is estimating the number of groups in the output of a group-by query on a base table, such as the following query on the Internet Movie Database (IMDb):

```
SELECT kind_id, phonetic_code, COUNT(*)
FROM title
GROUP BY kind_id, phonetic_code
```

Even for read-only OLAP workloads, this can be a difficult task. While it is trivial to store the number of distinct values of individual columns, it is challenging for multiple columns: That is, because the distinct value counts of individual columns or combinations of columns cannot easily be combined, and consequently, a system would have to maintain an exponential number of such counts to support arbitrary group-by expressions [10].

To work around multi-column statistics, systems like PostgreSQL use sampling to estimate the cardinality of such queries: They take a uniform sample from the relation, and extrapolate statistics obtained on this sample to the whole relation. However, as Charikar et al. proved in their seminal paper [5], it is fundamentally impossible to obtain good estimates from reasonably-sized samples, mainly due to possibly skewed data distributions. In other words, we would have to sample most of the relation in order to reliably achieve high estimation accuracy. As this is clearly not feasible, systems that employ sampling often have no choice but to use wildly inaccurate estimates for query optimization [26].

In previous work [10], we thus proposed a hybrid approach that combines single-column statistics with a sampling-based estimator to produce highly accurate multi-column estimates. However, while this approach achieves state-of-the-art estimates and can even deal with updates, we will show in this paper that it breaks down when there are selection predicates (filters) on the base table:

```
SELECT kind_id, phonetic_code, COUNT(*)
FROM title
WHERE production_year=2010
GROUP BY kind_id, phonetic_code
```

We will call queries like this *filtered group-by queries* in the following. To accurately predict such queries, the hybrid approach depends on precise estimates of the single-column cardinalities (i.e., `kind_id` and `phonetic_code`) *after* the selection predicate has been applied. Since statistics such as histograms cannot account for selections on dimensions that are not part of the statistic (`production_year` in this example), we must estimate these cardinalities purely based on information obtained from the sample. As outlined above, this will produce provably poor single-column estimates [5], which in turn severely deteriorate the overall accuracy of the estimator. In addition, as all sampling-based estimators, the hybrid approach suffers from 0-tuple situations where no samples remain after the predicate has been applied, leaving it with nothing but an "educated" guess about the number of groups.

Also, even if we would know the total number of groups (without a selection), selectivity estimation alone would not help. We need to estimate the number of *groups* in the result, whereas selectivity estimation can only accurately predict the number of *tuples* [24]. The presence of a selection has an non-trivial effect on the number of groups. For example, constraining `production_year` to 2010 as in the above query results in more than one third of the total number of groups, while setting it to 1990 only yields about 1% of the groups.

1st International Workshop on Applied AI for Database Systems and Applications (AIDB'19), August 26, 2019, Los Angeles, California, CA, USA.

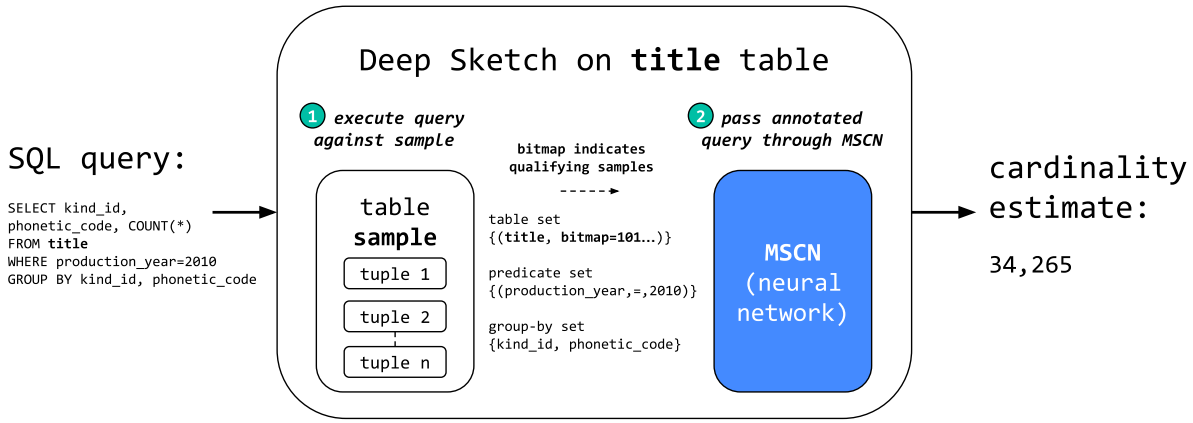

Figure 1: Querying a Deep Sketch.

Long story short, despite a large volume of previous research activity on the subject [5, 12], traditional approaches continue to struggle with producing accurate cardinality estimates on queries such as filtered group-by queries.

On the other hand, machine learning (ML) has recently been shown to be useful for a variety of system tasks, including classical problems like parameter tuning [3], query optimization [22, 28], and even indexing [21].

With Deep Sketches [18, 19], we have recently introduced a more robust approach to cardinality estimation that combines (supervised) ML with runtime sampling to capture skewed data distributions and correlations. A Deep Sketch contains a (set-based) deep learning model that is trained on a set of (generated) queries. First, these queries are generated using schema information and actual literals (values) from the database. Next, they are *annotated* with bitmaps indicating qualifying base table samples as well as their true cardinalities (*labels* in ML), obtained by executing these queries against the database. While Deep Sketches were primarily developed to estimate join queries [18], they can also be applied to the filtered group-by estimation task: The output cardinality of a join query is independent of the concrete query plan—e.g., both $(A \bowtie B) \bowtie C$ and $A \bowtie (B \bowtie C)$ can be represented as a set $\{A, B, C\}$. Similarly, the cardinality of a group-by query does not depend on the concrete permutation of the group-by columns in the SQL string. Thus, we can apply the same set-based model here.

Our original model (called multi-set convolutional network) represents three sets (tables, joins, and predicates) [18]. In this work, we use the original join module as a group-by module to estimate filtered group-by queries on base tables. Essentially, instead of encoding identifiers of join predicates, we now encode identifiers of group-by columns. We train the model with uniformly distributed queries on the IMDb `title` table. Our evaluation shows that Deep Sketches can effectively capture correlations between the group-by and the selection operation. We demonstrate that our approach significantly outperforms the estimators of PostgreSQL and HyPer[1] [16] as well as our own state-of-the-art multi-column estimator (hybrid approach). Like the state size of our hybrid approach, the size of our ML model only grows linearly with the number of columns while it can accurately predict the effect of selections. Our results indicate that deep learning is a promising candidate for the filtered group-by cardinality estimation task.

While our approach only supports the class of filtered group-by queries, its predictions can of course also be used to estimate (sub trees of) larger queries. Another important application is hash table sizing for hash-based aggregations. Also, in distributed query processing, where it is expensive to re-optimize at query runtime (as advocated by [35]), accurately estimating such queries upfront is of high value. For example, it allows the optimizer to decide between a local and a shuffle-based aggregation depending on whether there are few or many unique values, respectively.

Note that Deep Sketches are currently limited to read-only databases and would require complete re-training (or at least incremental training) to support updates. Also, our model currently only supports conjunctive equality and range predicates. We refer the reader to [18] for a discussion on these two topics.

Contributions. In summary, we make the following contributions:

- We experimentally show the poor performance of state-of-the-art approaches for estimating filtered group-by queries.
- We adapt an approach that combines deep learning with runtime sampling to this problem, and show that it provides higher accuracy with linear state size.
- We discuss possible extensions to our approach that address its limitations: expensive training due to large query space, limited to filtered group-by queries.

2. RELATED WORK

Traditional Approaches. Accurate cardinality estimates are a fundamental requirement for effective query optimization [24], and, correspondingly, a plethora of approaches has been proposed in the literature. Traditionally, systems maintain some sort of statistics on base tables, and attempt to derive cardinality estimates from these statistics [24, 31]. For this purpose, sampling is one of the most versatile approaches, as it offers attractive performance and naturally

[1]We are referring to the research version of HyPer developed at the Technical University of Munich.

deals with selections and multi-column estimates. However, as Charikar et al. proved, any exclusively sampling-based approach must have poor accuracy on some input data [5]. A large body of work thus attempts to improve sampling-based estimation with auxiliary information [9, 11, 23, 25, 44]. This includes our own state-of-the-art hybrid estimator, which, as opposed to previous approaches, can produce accurate multi-column estimates with minimal overhead but struggles in the presence of selections [10].

Current systems frequently assume that the individual attributes are independent for multi-column estimation, and use ad-hoc heuristics to estimate the impact of selections [24]. However, such heuristics frequently do not reflect real-world data accurately, resulting in large estimation errors [42]. Multi-column statistics, for example multi-dimensional histograms [36], or wavelets [6], promise more accurate cardinality estimates, but as opposed to Deep Sketches, their space consumption is non-linear in the number of attributes.

ML for Databases. Recently, there has been high interest in applying ML techniques to database problems. The vision is to enhance or even replace traditional database components with ML counterparts to create a learned database system [20]. Recent work in the query optimization domain focuses on join enumeration [22, 29], plan rewriting [7], and even proposes an end-to-end learned optimizer [28]. As opposed to these proposals, we advocate for a more gradual approach that focuses on cardinality estimation and leaves join enumeration to modern algorithms that can find the optimal join order for dozens of relations [33]. Given the highly non-trivial nature of the cardinality estimation problem and the poor performance of current implementations [24], we believe that ML can make a large difference here. Previous ML papers on this problem used pure ML without any runtime features [34, 40]. In previous work, we have started to explore how ML can be combined with runtime sampling to estimate base table and join cardinalities [18, 19]. In this work, we adapt this approach to filtered group-by queries.

Other recent work proposes learned components for adaptive query processing [39], classical [21] and succinct [41] index structures, view materialization [27], index tuning [4, 8], data partitioning [14], workload management [15], and query performance prediction [30].

Unsupervised ML Approaches. Unsupervised ML has not yet been adapted to group-by queries. The closest work is [13] in which the authors propose to use an autoregressive model to learn a conditional probability distribution. However, their approach only supports equality predicates and does not address group-by queries. Another recent proposal is to use deep generative models to capture the joint data distributions of multiple attributes and to generate new sample tuples following that distribution [38]. The idea then is to run actual queries on these representative samples. Besides its application in approximate query processing, one could use that approach to estimate cardinalities of filtered group-by queries. However, since the sample tuples are generated from the joint distribution over all columns, selective queries may require a high number of samples to achieve good approximation performance.

3. DEEP LEARNING TO THE RESCUE

In this section, we describe how we are using deep learning to accurately predict filtered group-by queries on base tables. In our approach, we utilize Deep Sketches [18, 19] which combine (supervised) ML with runtime sampling to capture data skew and correlations. A Deep Sketch consists of a trained (set-based) neural network and a set of materialized samples, typically containing 1,000 base table tuples selected uniformly at random.

Figure 1 shows the query flow of a Deep Sketch. First (1), the query's selection predicates are evaluated on the table sample to obtain a bitmap indicating qualifying samples (we call this process *query annotation*), before (2) the annotated query is featurized and passed through a neural network that outputs the query's estimated cardinality.

While Deep Sketches were primarily developed to estimate join queries [18], they can also be used to estimate (filtered) group-by queries as we show in this paper. Similar to a join query where the output cardinality is independent of the concrete query plan, the cardinality of a group-by query does not depend on the concrete permutation of the group-by columns in the SQL string. Thus, we can apply the same set-based model here.

In the following, we provide a high-level description of our model for the purpose of this paper. For a more detailed description of the model architecture and the query featurization, we refer the reader to [18].

Query Featurization. Like in our initial work [18], we represent a query $q \in Q$ as a collection (T_q, G_q, P_q) of a set of tables $T_q \subset T$, a set of group-by columns $G_q \subset G$ and a set of (conjunctive) predicates $P_q \subset P$ participating in the specific query q (illustrated in the middle of Figure 1). The only difference here is that we have replaced the original join set with a group-by set, since this work focuses on filtered group-by queries on base tables and does not consider joins. We later outline in Section 5 how to also address joins. T, G, and P describe the entire query space in terms of available tables, group-by columns, and predicates, respectively. Note that for the purpose of this work T_q always only consists of a single table.

All categorical data, i.e., tables, columns, and predicate types ($=$, $<$, and $>$), are enumerated based on their appearance in the training data and represented as unique one-hot vectors (binary vectors with a single non-zero entry). Bitmaps that indicate qualifying samples are represented in their natural form as binary vectors, typically of length 1,000 with each bit representing a table sample. Non-zero bits indicate that the corresponding sample qualifies all predicates. Query literals are normalized (mapped to the range $[0, 1]$) using the minimum and maximum values of the respective column. Cardinalities are first logarithmized (to obtain a more even distribution) and then normalized using the maximum cardinality present in the training data.

Neural Network Architecture. Our multi-set convolutional network (MSCN) model [18] (code: [1]) represents sets of tables, joins, and predicates in separate modules. The architecture of each of these modules is inspired by *Deep Sets* [43], a neural network module that operates on sets. While the Deep Sets model only addresses single sets, the MSCN model represents multiple sets in a single architecture and can capture correlations between sets. Each module is comprised of one two-layer fully-connected multi-layer perceptron (MLP) per set element with shared parameters. We average module outputs, concatenate them, and pass them through a final output MLP, which captures correlations

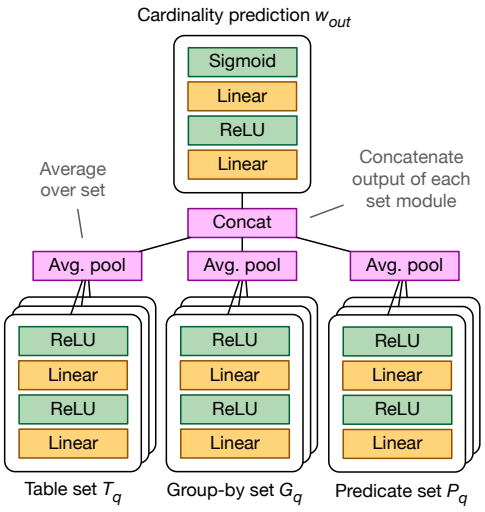

Figure 2: Architecture of our MSCN model. Tables, group-by columns, and predicates are represented as separate MLP modules that provide input to a final output network that predicts query cardinalities.

between sets and outputs a cardinality estimate. Since we apply a learnable mapping for each set element individually (with shared parameters), which is similar to the concept of a 1×1 convolution, often used in CNNs for image classification [37], we call our model convolutional.

In this work, we use the join module as a group-by module to estimate filtered group-by queries on base tables. Figure 2 illustrates the (new) model that operates on the query representation (T_q, G_q, P_q) as follows:

$$\text{Table module:} \quad w_T = \frac{1}{|T_q|} \sum_{t \in T_q} \text{MLP}_T(v_t)$$

$$\text{Group-by module:} \quad w_G = \frac{1}{|G_q|} \sum_{g \in G_q} \text{MLP}_G(v_g)$$

$$\text{Predicate module:} \quad w_P = \frac{1}{|P_q|} \sum_{p \in P_q} \text{MLP}_P(v_p)$$

$$\text{Merge \& predict:} \quad w_{\text{out}} = \text{MLP}_{\text{out}}([w_T, w_G, w_P])$$

The advantage of this architecture over standard neural network architectures such as simple MLPs is that it does not require *serialization* of the data. In other words, we do not need to convert the individual set elements to an ordered sequence. Instead, for every set S, we learn a set-specific, per-element neural network $\text{MLP}_S(v_s)$ that is applied on every feature vector v_s for every element $s \in S$ individually. We then derive a final representation w_S for this set by *averaging* these individual transformed representations, i.e., $w_S = 1/|S| \sum_{s \in S} \text{MLP}_S(v_s)$. Note that an average of (transformed) one-hot vectors uniquely identifies the combination of one-hot vectors, e.g., the set of group-by columns participating in the query. Thus, our architecture does not waste any capacity in learning to discover the symmetries and structure of the original representation, such as the boundaries between sets of different size. The individual set representations are then concatenated and passed through a final output MLP: $w_{\text{out}} = \text{MLP}_{\text{out}}([w_{S_1}, w_{S_2}, \ldots, w_{S_N}])$, where N is the total number of sets and $[\cdot, \cdot]$ denotes vector concatenation. All MLP modules use $\text{ReLU}(x) = \max(0, x)$ activation functions and the output MLP uses a $\text{sigmoid}(x) = 1/(1 + \exp(-x))$ activation function for the last layer to only output a scalar value $w_{\text{out}} \in [0, 1]$.

column	cardinality	filter	group-by
kind_id	6		✓
phonetic_code	23259		✓
episode_nr	14907	✓	✓
production_year	133	✓	✓

Table 1: Cardinalities (distinct value counts) of columns used in our workload. Checkmarks indicate whether columns can appear in filter and/or group-by clauses.

Optimization Metric. We make use of the Adam [17] optimizer to minimize the mean *q-error* [32] q ($q \geq 1$). The q-error is the factor between the true and the estimated cardinality.

Query Generation and Annotation. We obtain an initial training dataset by generating random queries using schema information and actual literals (values) from the database. The query generator is parametrized with a set of available group-by and selection columns as well as predicate types ($=$, $<$, and $>$). In addition, we specify limits on the number of group-by and selection columns. All decisions are performed uniformly at random (e.g., how many and which group-by columns a specific query contains). We provide a concrete example of this process in our evaluation in Section 4.

Once the queries have been generated, we execute them against a set of materialized sample tuples to annotate them with bitmaps indicating qualifying samples and against the full database table to obtain their true cardinalities.

4. EVALUATION

Dataset and Query Workload. We use the real-world IMDb dataset, which is challenging for cardinality estimators [24] due to data skew and correlations. We focus our evaluation on the `title` table, which contains more than 2.5 M movie titles produced over 133 years.

In the following, we describe our query workload including the search space (space of possible queries) that it is drawn from.

We generate a workload that operates on four columns (cf. Table 1). The construction of a query works as follows (all decisions are performed uniformly at random): We generate either one or two filter predicates (selections) on the filter columns, i.e., either one filter (on `episode_nr` *or* on `production_year`) or two filters (on `episode_nr` *and* on `production_year`). Each filter can be an equality ($=$), a less than ($<$), or a greater than ($>$) predicate. Filter literals are drawn from the respective columns in the database (e.g., 2010 for a filter on `production_year`). In fact, we pick values from random (uniformly distributed) row offsets. Next, we generate the group-by clause. We generate either one or two group-by columns, chosen among all four columns (without replacement).

Thus, given the distinct value counts shown in Table 1, there are 5,993,013 possible selections and 10 possible group-by clauses, resulting in 59,930,130 possible *unique* queries.

For our query workload, we use 500 queries from within that search space.

Deep Sketch. We use a Deep Sketch (DS) that is trained with uniformly distributed queries within the above query space. The training set is *disjoint* from the query workload. By default, we use 10,000 training queries and 1,000 materialized samples (i.e., each query is annotated with 1,000 bits).

We use the MSCN model with the following hyperparameters: 100 epochs, batch size of 1024, 256 hidden units, and a learning rate of 0.001. Training the model with 10,000 queries takes around two minutes on a GeForce GTX 1050 Ti GPU using the PyTorch framework [2] with CUDA. Querying the model with 500 annotated queries takes around two microseconds per query when running the model on the GPU. When serialized to disk, the model itself consumes 2.5 MiB (and another 16 KiB are occupied by the table sample). Note that its size increases linearly with the number of columns: Since columns are encoded as unique one-hot vectors, we require one additional 256-dimensional vector (# of hidden units) for each extra column.

Competitors. We compare against PostgreSQL version 10.3 and HyPer as well as our state-of-the-art multi-column estimator SCBC [10]. At its core, SCBC employs an improved sampling-based estimator derived from the estimators proposed by Charikar et al. [5]. For our experiments, this part of SCBC uses the same sample than the Deep Sketch. As outlined previously, a sampling-only approach will have provably poor accuracy in some cases. For this reason, SCBC additionally maintains the distinct value counts in the individual columns. In order to obtain a multi-column estimate, SCBC computes lower and upper bounds on the number of groups based on these single-column counts, which are then used to constrain the output of the sampling-based estimator.

Without any selections, this approach gives excellent estimates due to the accurate single-column estimates. In order to extend SCBC to filtered group-by queries, we can simply execute the query on the sample, and run the sampling-based part of SCBC afterwards. However, we cannot predict the impact of selections on the number of distinct values in the individual columns accurately. This prevents SCBC from computing a reliable lower bound on the number of groups after selections. For the purposes of this paper, we only compute an upper bound on the number of groups based on the single-column cardinalities *before* any selections. Note that for very selective queries, this bound will be much too loose, severely limiting the performance of SCBC.

During our preliminary evaluation, we have tried to compute single-column estimates after selections. However, without any additional information, this boils down to a sample-based distinct value count estimation, which gives poor results [5]. As SCBC heavily relies on accurate single-column estimates, such an approach cannot improve the accuracy of SCBC on filtered group-by queries.

Estimation Quality. Figure 3 and Table 2 show the q-errors of the different approaches. While PostgreSQL and HyPer achieve reasonable median q-errors, they both have heavy outliers. SCBC has a lower median q-error than PostgreSQL and HyPer, but also suffers from large mispredictions. These mispredictions arise mainly due to the presence of selections in the workload. As outlined above, selections prevent SCBC from computing tight bounds on the result cardinality, and there are many cases in which it has to rely purely on the potentially inaccurate sampling-based estimator. DS achieves the best median estimation quality and is significantly more robust than its competitors. In fact, the estimates produced by DS are within a factor of 3 of the true cardinalities on average. Note that DS only observed 10,000 of the almost 60 M possible queries during training.

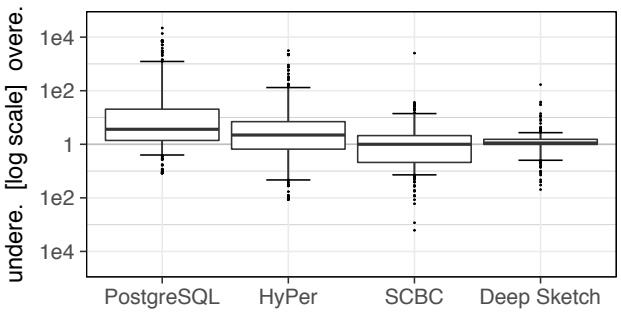

Figure 3: Estimation errors on the query workload. The box boundaries are at the 25th/75th percentiles and the horizontal "whisker" lines mark the 95th percentiles.

	median	90th	95th	99th	max	mean
PostgreSQL	4.35	318	1236	7431	21934	351
HyPer	6.00	111	131	2275	3144	74.1
SCBC	3.21	14.0	21.0	86.8	2528	16.6
Deep Sketch	**1.31**	**3.00**	**5.12**	**30.0**	**169**	**2.58**

Table 2: Estimation errors on the query workload.

Effect of Number of Training Queries. We now analyze the performance of DS with fewer training queries. Figure 4 shows the q-errors of DS with a varying number of training queries. With only 100 queries, DS already achieves a median q-error of 4.33 which is competitive with the other approaches. However, as one would expect with such a small training dataset, it produces large outliers. Starting with 1,000 training queries, DS surpasses HyPer in the 95th percentile. With 5,000 queries, it eventually outperforms SCBC in all metrics.

Effect of Selections. We now use another (manually constructed) query workload to highlight the effect of selections. Like in our opening example, we have an equality predicate on `production_year` (and vary the literal from 1880 to 2020) and use the `kind_id` and `phonetic_code` columns in the group-by clause.

The Deep Sketch is again trained using the same 10,000 queries as above.

Here, we also include a simple combinatorial estimation approach (which we call *baseline*) that has access to the total number of groups G and the true number of qualifying tuples t. Note that in reality we do not know the exact values of these variables. This approach assumes a uniform distribution of values among groups and works as follows:

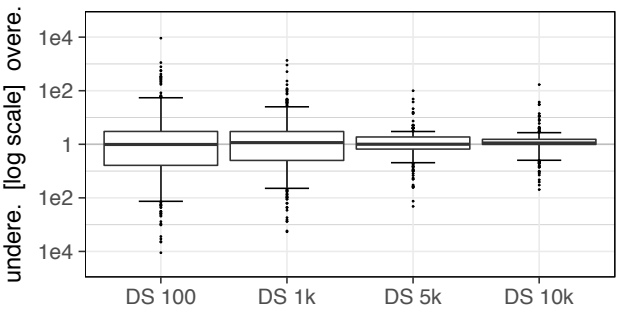

Figure 4: Estimation errors on the query workload with an increasing number of training queries.

Assume the relation contains G groups and $N = G \times n$ tuples with n being the number of tuples per group. A query selects t tuples randomly without replacement, i.e., we disregard a possible correlation between the selection column and the group-by column. Then the expected number of groups in the result set is:

$$G \times \left(1 - \frac{\binom{N-n}{t}}{\binom{N}{t}}\right)$$

Figure 5 shows the results with the four approaches in different columns. The X axis denotes the production year while we plot the cardinalities on the Y axis. The green line shows the true cardinalities, while the red line denotes the estimated cardinalities. In addition, the *dotted* red line shows the estimates of the baseline approach.

We first only group by the low cardinality column `kind_id` (first row in Figure 5). As it turns out (green line), the number of different movie kinds increases over time. For example, in 1931 the first *tv series* came out. Before that year, all titles were of kind *movie*, *tv movie*, or *episode*. Our baseline (dotted red line) overestimates the number of distinct movie kinds early on. It essentially assumes that the number of distinct movie kinds directly correlates with the number of movie titles per year. In fact, once the number of yearly movie titles increases (which it actually does), the baseline estimates more distinct movie kinds. The reason for its overestimation is that it assumes that the qualifying movie titles t are uniformly distributed among movie kinds. In reality, however, this distribution is non-uniform and highly correlated with the year. Independent of the year, PostgreSQL and HyPer always estimate 6 distinct movie kinds, which is the total number of distinct values (cf. Table 1). SCBC's estimates fluctuate between 1 and 5. This result further illustrates the importance of tight bounds on the output cardinality for SCBC. The predicates in this workload are generally quite selective, i.e., the number of groups in the output is small. However, SCBC only has access to the single-column cardinalities *before* these selections, which provide no useful upper bound on the number of groups in the output. Thus, SCBC almost exclusively relies on its sampling-based estimator, which leads to the observed severe overestimations.

Moreover, SCBC also suffers heavily from 0-tuple situations, which explains the observed frequent underestimations. Our query workload contains 132 different equality predicates on `production_year`, whereas the sample used by SCBC only contains 100 distinct values of `production_year`. Thus, there are 32 queries in which the sample will contain no tuples after applying the selection, and SCBC can only guess the output cardinality. Since no lower bound on the number of groups in the output is available, SCBC can also not correct this guess reasonably. As Figure 5 illustrates, DS does not suffer from this problem.

In order to confirm these considerations, we ran the same workload but gave SCBC access to the true column cardinalities *after* selections. As expected, this enabled SCBC to predict the result cardinalities with high accuracy.

DS captures the increase in movie kinds but still underestimates it. The reason is that certain kinds are rare and are thus less likely to be captured during training.

Next, we use the high cardinality column `phonetic_code` in the group-by clause (second row in Figure 5). Again, the number of distinct values essentially increases over time (green line), with a sharp rise after 1980. Our baseline (dotted red line) produces reasonable estimates up to the year 1940 before starting to overestimate the cardinality in recent years. The reason is again that it assumes a direct correlation between the number of yearly movie titles and the number of distinct phonetic codes. PostgreSQL's and HyPer's estimates are again hardly unaffected by the selection. While PostgreSQL catches the increase in distinct values in recent years, HyPer heavily underestimates all of these queries. SCBC's estimates again fluctuate due to the same reasons outlined above. DS estimates closely approximate the true cardinalities with a few exceptions.

Finally, we group by both `kind_id` and `phonetic_code` (third row in Figure 5). The trend (green line) is similar than for the queries that only group-by `phonetic_code` (second row), but interestingly the estimations of PostgreSQL, HyPer, and SCBC differ: PostgreSQL estimates the spike in recent years to be larger this time, while HyPer's estimates are now influenced by the selection. SCBC's estimates still fluctuate but are now much closer to the true cardinalities, except in recent years where its sampling-based estimator underestimates the number of groups. DS again produces highly accurate estimates with minimal outliers.

In summary, all of the traditional approaches have issues with this workload. DS, in contrast, captures the correlations between the selection on `production_year` and the group-by columns, leading to more accurate estimates.

5. DISCUSSION

More Runtime Features. Currently, we only featurize bitmaps indicating table samples that have survived the selection. Since we need to consult the sample anyways, we could also derive and encode further statistics, such as the number of distinct values of individual group-by columns in the sample *before* and *after* selection, normalized based on the number of distinct values in the entire table. Similarly, we could compute and encode the number of groups in the sample *before* and *after* selection (again normalized based on table statistics).

Combining DS and SCBC. While we have shown that DS only needs to observe 0.02% of the query space to produce highly accurate estimates for our query workload, this might not always be the case. To prune the search space and thus reduce the number of required training queries, we could combine DS with SCBC. We have previously shown that

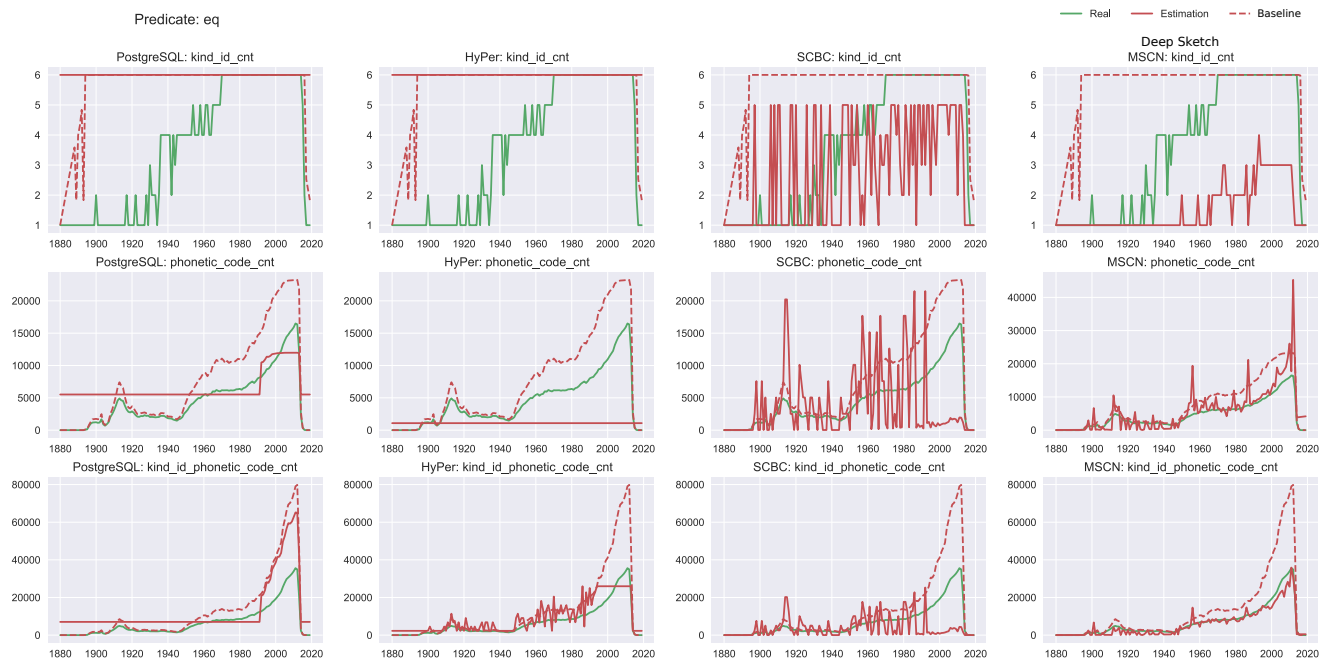

Figure 5: Cardinality estimates (Y axis) of PostgreSQL, HyPer, SCBC, and Deep Sketches. Selection on `production_year` (X axis) and group-by on `kind_id` and/or `phonetic_code`.

SCBC produces accurate group-by estimates *without* selections when it has access to precise estimates of the number of distinct values of individual columns [10]. Under selections, SCBC does not have such estimates and as Charikar et al. proved [5], also cannot compute precise estimates based on a sample. To provide SCBC with precise estimates of single-column distinct value counts *after* selection, we could train a DS specifically for this task. By focusing on single group-by columns, we would effectively prune the search space and would need fewer training queries. The downside is that a combined DS/SCBC approach would still suffer from 0-tuple situations, i.e., when the sampling-based estimator of SCBC has no qualifying tuples to start with. DS on its own does not have this problem and can still rely on static query features (i.e., group-by columns and predicates) in such cases [18].

Supporting Joins. In the current model, we have replaced the original join module with a group-by module. Instead, we could add a forth set module to capture group-by and join queries. While this is a rather straightforward extension, this would increase the query space and thus require a larger training dataset.

6. CONCLUSIONS

We have experimentally shown that traditional query optimizers as well as state-of-the-art techniques for distinct value estimation produce poor cardinality estimates for filtered group-by queries. We have extended and evaluated an approach that combines a (set-based) deep learning model with runtime sampling to accurately predict such queries. This approach not only features higher accuracy but also linear state size. We therefore believe that deep learning is a promising candidate for solving the filtered group-by cardinality estimation task.

7. REFERENCES

[1] Learned Cardinalities in PyTorch. `https://github.com/andreaskipf/learnedcardinalities`.

[2] PyTorch. `https://pytorch.org/`.

[3] D. V. Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic Database Management System Tuning Through Large-scale Machine Learning. In *SIGMOD*, pages 1009–1024, 2017.

[4] J. Arulraj, R. Xian, L. Ma, and A. Pavlo. Predictive Indexing. *CoRR*, abs/1901.07064, 2019.

[5] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards Estimation Error Guarantees for Distinct Values. In *PODS*, pages 268–279, 2000.

[6] G. Cormode, M. N. Garofalakis, P. J. Haas, and C. Jermaine. Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches. *Foundations and Trends in Databases*, 4(1-3):1–294, 2012.

[7] G. Damasio, V. Corvinelli, P. Godfrey, P. Mierzejewski, A. Mihaylov, J. Szlichta, and C. Zuzarte. Guided Automated Learning for query workload re-Optimization. *CoRR*, abs/1901.02049, 2019.

[8] B. Ding, S. Das, R. Marcus, W. Wu, S. Chaudhuri, and V. Narasayya. AI Meets AI: Leveraging Query Executions to Improve Index Recommendations. May 2019.

[9] B. Ding, S. Huang, S. Chaudhuri, K. Chakrabarti, and C. Wang. Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee. In *SIGMOD*, pages 679–694, 2016.

[10] M. Freitag and T. Neumann. Every Row Counts: Combining Sketches and Sampling for Accurate Group-By Result Estimates. In *CIDR*, 2019.

[11] P. B. Gibbons. Distinct Sampling for Highly-Accurate Answers to Distinct Values Queries and Event Reports. In *VLDB*, pages 541–550, 2001.

[12] H. Harmouch and F. Naumann. Cardinality Estimation: An Experimental Survey. *PVLDB*, 11(4):499–512, 2017.

[13] S. Hasan, S. Thirumuruganathan, J. Augustine, N. Koudas, and G. Das. Multi-Attribute Selectivity Estimation Using Deep Learning. *CoRR*, abs/1903.09999, 2019.

[14] B. Hilprecht, C. Binnig, and U. Röhm. Learning a Partitioning Advisor with Deep Reinforcement Learning. *CoRR*, abs/1904.01279, 2019.

[15] S. Jain, J. Yan, T. Cruanes, and B. Howe. Database-Agnostic Workload Management. In *CIDR*, 2019.

[16] A. Kemper and T. Neumann. HyPer: A Hybrid OLTP & OLAP Main Memory Database System Based on Virtual Memory Snapshots. In *ICDE*, pages 195–206, 2011.

[17] D. P. Kingma and J. Ba. Adam: A Method for Stochastic Optimization. In *ICLR*, 2015.

[18] 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.

[19] A. Kipf, D. Vorona, J. Müller, T. Kipf, B. Radke, V. Leis, P. Boncz, T. Neumann, and A. Kemper. Estimating Cardinalities with Deep Sketches. *arXiv preprint arXiv:1904.08223*, 2019.

[20] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. SageDB: A Learned Database System. In *CIDR*, 2019.

[21] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The Case for Learned Index Structures. In *SIGMOD*, pages 489–504, 2018.

[22] S. Krishnan, Z. Yang, K. Goldberg, J. M. Hellerstein, and I. Stoica. Learning to Optimize Join Queries With Deep Reinforcement Learning. *CoRR*, abs/1808.03196, 2018.

[23] P. Larson, W. Lehner, J. Zhou, and P. Zabback. Cardinality Estimation Using Sample Views with Quality Assurance. In *SIGMOD*, pages 175–186, 2007.

[24] V. Leis, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. How Good Are Query Optimizers, Really? *PVLDB*, 9(3):204–215, 2015.

[25] V. Leis, B. Radke, A. Gubichev, A. Kemper, and T. Neumann. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*, 2017.

[26] V. Leis, B. Radke, A. Gubichev, A. Mirchev, P. A. Boncz, A. Kemper, and T. Neumann. Query Optimization Through the Looking Glass, and What We Found Running the Join Order Benchmark. *VLDBJ*, 27(5):643–668, 2018.

[27] X. Liang, A. J. Elmore, and S. Krishnan. Opportunistic View Materialization with Deep Reinforcement Learning. *CoRR*, abs/1903.01363, 2019.

[28] R. Marcus, P. Negi, H. Mao, C. Zhang, M. Alizadeh, T. Kraska, O. Papaemmanouil, and N. Tatbul. Neo: A Learned Query Optimizer. *arXiv preprint arXiv:1904.03711*, 2019.

[29] R. Marcus and O. Papaemmanouil. Deep Reinforcement Learning for Join Order Enumeration. In *aiDM*, pages 3:1–3:4, 2018.

[30] R. Marcus and O. Papaemmanouil. Plan-Structured Deep Neural Network Models for Query Performance Prediction. *CoRR*, abs/1902.00132, 2019.

[31] A. Metwally, D. Agrawal, and A. El Abbadi. Why Go Logarithmic if We Can Go Linear? Towards Effective Distinct Counting of Search Traffic. In *EDBT*, pages 618–629, 2008.

[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] T. Neumann and B. Radke. Adaptive Optimization of Very Large Join Queries. In *SIGMOD*, pages 677–692, 2018.

[34] J. Ortiz, M. Balazinska, J. Gehrke, and S. S. Keerthi. Learning State Representations for Query Optimization with Deep Reinforcement Learning. In *DEEM*, pages 4:1–4:4, 2018.

[35] M. Perron, Z. Shang, T. Kraska, and M. Stonebraker. How I Learned to Stop Worrying and Love Re-optimization. *arXiv preprint arXiv:1902.08291*, 2019.

[36] M. Shekelyan, A. Dignös, and J. Gamper. DigitHist: A Histogram-Based Data Summary with Tight Error Bounds. *PVLDB*, 10(11):1514–1525, 2017.

[37] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the Inception Architecture for Computer Vision. In *CVPR*, 2017.

[38] S. Thirumuruganathan, S. Hasan, N. Koudas, and G. Das. Approximate Query Processing using Deep Generative Models. *arXiv preprint arXiv:1903.10000*, 2019.

[39] I. Trummer, S. Moseley, D. Maram, S. Jo, and J. Antonakakis. SkinnerDB: Regret-Bounded Query Evaluation via Reinforcement Learning. *PVLDB*, 11(12):2074–2077, 2018.

[40] 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.

[41] Y. Wu, J. Yu, Y. Tian, R. Sidle, and R. Barber. Designing Succinct Secondary Indexing Mechanism by Exploiting Column Correlations. *CoRR*, abs/1903.11203, 2019.

[42] X. Yu, C. Zuzarte, and K. C. Sevcik. Towards Estimating the Number of Distinct Value Combinations for a Set of Attributes. In *CIKM*, pages 656–663, 2005.

[43] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Póczos, R. R. Salakhutdinov, and A. J. Smola. Deep Sets. In *NIPS*, pages 3394–3404, 2017.

[44] M. Zaït, S. Chakkappen, S. Budalakoti, S. R. Valluri, R. Krishnamachari, and A. Wood. Adaptive Statistics in Oracle 12c. *PVLDB*, 10(12):1813–1824, 2017.