FACE: A Normalizing Flow based Cardinality Estimator

Jiayi Wang, Chengliang Chai, Jiabin Liu, Guoliang Li
Department of Computer Science, Tsinghua University, China
{jiayi-wa20@mails.,ccl@,liujb19@mails.,liguoliang@}tsinghua.edu.cn

ABSTRACT

Cardinality estimation is one of the most important problems in query optimization. Recently, machine learning based techniques have been proposed to effectively estimate cardinality, which can be broadly classified into query-driven and data-driven approaches. Query-driven approaches learn a regression model from a query to its cardinality; while data-driven approaches learn a distribution of tuples, select some samples that satisfy a SQL query, and use the data distributions of these selected tuples to estimate the cardinality of the SQL query. As query-driven methods rely on training queries, the estimation quality is not reliable when there are no high-quality training queries; while data-driven methods have no such limitation and have high adaptivity.

In this work, we focus on data-driven methods. A good data-driven model should achieve three optimization goals. First, the model needs to capture data dependencies between columns and support large domain sizes (achieving high accuracy). Second, the model should achieve high inference efficiency, because many data samples are needed to estimate the cardinality (achieving low inference latency). Third, the model should not be too large (achieving a small model size). However, existing data-driven methods cannot simultaneously optimize the three goals. To address the limitations, we propose a novel cardinality estimator FACE, which leverages the Normalizing Flow based model to learn a continuous joint distribution for relational data. FACE can transform a complex distribution over continuous random variables into a simple distribution (e.g., multivariate normal distribution), and use the probability density to estimate the cardinality. First, we design a dequantization method to make data more "continuous". Second, we propose encoding and indexing techniques to handle Like predicates for string data. Third, we propose a Monte Carlo method to efficiently estimate the cardinality. Experimental results show that our method significantly outperforms existing approaches in terms of *estimation accuracy* while keeping similar *latency* and *model size*.

1 INTRODUCTION

Cardinality estimation (CE) is a fundamental and significant problem that has been widely studied for many years. It aims to estimate the number of records that satisfy a given query in a database. CE
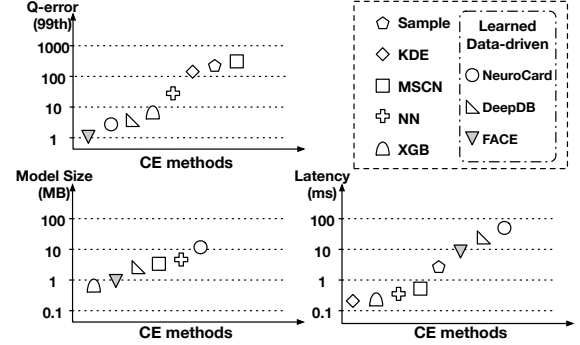
* Chengliang Chai and Guoliang Li are corresponding authors.

Figure 1: Performance comparison of CE methods.

has widespread applications in the database community, such as query optimization, approximate query processing, query profiling, etc. Especially, a precise CE approach directly influences the quality of the optimized query plan, leading to orders of magnitude performance improvement. Since traditional methods, e.g., histograms [35], sampling [21, 47] or kernel density based methods [10, 15], cannot capture the column correlations, recently machine learning (ML) based CE methods [6, 11, 17, 23–27, 37–39, 41, 44–46, 48] have been proposed, which can achieve superior performance, because they have high representation capability and strong learning ability.

Generally speaking, a good learning-based CE model should achieve the following optimization objectives.

High accuracy (O1): The estimated cardinality should be close to the real cardinality, so as to obtain an optimized query plan, and the generalization ability is also important.

Low latency (O2): During a query plan generation, the CE module has to be triggered multiple times, so its latency is very important to generate an optimized plan efficiently.

Lightweight model size (O3): Considering the memory limitation, the model should not be large [44, 49], because a database has many schemas and requires to train a model for each schema. Moreover, a lightweight model can achieve high inference efficiency.

To achieve these optimization goals, *query-driven* and *data-driven* learned models have been proposed. The former [17, 37] learns a regression model that learns a mapping from a query to its cardinality. However, this approach relies on training queries and has a limited generalization ability on query changes and data changes. For example, if the training workload is different from the test workload, the performance is not reliable. Data-driven [11, 44, 45] approaches learn the joint distribution of data in a relational table without the query workload, and use the distribution to infer the cardinality. They do not need to know the query workload in advance and can generalize to unseen queries, and thus the generalization ability of data-driven methods is stronger than the query-driven ones (but they cannot adapt to data changes).

However, existing data-driven methods suffer from the following limitations. (1) Sum-product-network-based method [11] assumes different levels of independence between columns, based on which they recursively split rows and columns and estimate the cardinality using the sum-product network, but the accuracy is low due to the assumption (cannot achieve **O1**). Therefore, the first challenge is how to capture the dependencies between different columns (**C1**). (2) Although Naru [44, 45] and DQM-D [9] can leverage the auto-regressive model to capture dependencies by factorizing the joint distribution into conditional probability distributions, they cannot handle the table with a large domain size well, where the large domain size means that in the table there exist attributes with a large number of distinct cell values. Since the number of model parameters scales with the domain size [9, 45], it leads to high training cost and high storage overhead (cannot achieve **O3**). Even if NeuroCard [44] can alleviate this problem by dividing the column with the large domain size into multiple sub-columns, it sacrifices the accuracy (cannot achieve **O1**).

Besides, existing data-driven methods cannot efficiently support Like predicates on string data, because *i*) strings naturally have large domain size, and *ii*) for inference, it is slow to find strings satisfying the predicates (cannot achieve **O2**). Hence, how to support large domain size (including string data) while keeping high accuracy is the second challenge (**C2**). (3) In the inference step, for range queries, most data-driven methods [9, 44, 45] need to sample data points from the ranges, feed them into the trained model and use the inferred results to estimate the cardinality. This step is inefficient because it has to trigger the model inference many times for estimation (cannot achieve **O2**). Therefore, how to reduce the latency of the inference step is the third challenge (**C3**).

To address these challenges, we propose a Normalizing Flow based Cardinality Estimator, FACE, which approximates the joint distribution using the Normalizing Flow (NF) model. NF is a generative model that learns the joint probability distribution of data points. It [19, 30] consists of a sequence of invertible and differentiable transforms and can transform a complex distribution over *continuous* random variables into a simple distribution (e.g., multivariate normal distribution), and vice versa. So the probability density of each tuple can be computed. Intuitively, the term "Flow" refers to the trajectory that the data is gradually transformed by the sequence of transformations. The term "normalizing" refers to the fact that these data points are mapped into a simple distribution, usually multivariate normal distribution. As shown in Fig. 1, FACE shows superiority on all dimensions, and the reasons are as follows.

In general, since NF regards all columns in the table as a whole without any decomposition during training and inference, it can capture the dependencies of columns (addressing **C1**, for **O1**). First, as NF is adequate for modeling continuous data, it naturally can be utilized to handle large domain size data without expensive embeddings (addressing **C2**, for **O3**). Second, for discrete data (e.g., categorical data), we propose a dequantization technique to make them more "continuous", so as to fit the NF model and obtain accurate estimation (for **O1**). Third, we propose an effective method to encode string data, transform Like predicates to range ones and efficiently search qualified strings (for **O2**). Finally, we propose to leverage the query similarity to accelerate the inference (addressing **C3**, for **O2**). In summary, we make the following contributions.

(1) We propose a Normalizing Flow based framework that can efficiently and effectively address the CE problem.

(2) We propose a dequantization technique to handle discrete data, and design a string data encoding method to support strings.

(3) We leverage the query similarity to accelerate the inference.

(4) Experimental results showed that our method significantly outperformed existing approaches.

2 PRELIMINARY
2.1 Problem Definition
Consider a relation T with N tuples and m attributes $\{A_1, A_2, \cdots A_m\}$. Each tuple $t \in T$ is $t = (a_1, a_2, \cdots, a_m)$, where a_i is a cell value in A_i, $i = 1, \cdots, m$. $o(t)$ denotes the number of occurrences of t. The task of cardinality estimation (CE) is to estimate the result size without actually executing the query. The predicate θ of the query can be viewed as a function that takes as input t, and outputs $\theta(t) = 1$ if t satisfies the predicate, otherwise $\theta(t) = 0$. Hence, the cardinality can be formally defined as $car(\theta) = |\{t \in T : \theta(t) = 1\}|$, and the selectivity of θ is denoted by $sel(\theta) = car(\theta)/N$.

Note that $sel(\theta)$ can be computed using the joint data distribution over the attribute domains in T [45]:

$$sel(\theta) = \sum_{t \in A_1 \times \cdots \times A_m} \theta(t) \cdot P(t) \tag{1}$$

where $P(t) = o(t)/n$ denotes the probability of tuple t. Thus one can estimate $car(\theta)$ by computing the probability distribution.
Supported Query Predicate. In this part, we show the predicates of queries that we can support for CE. (1) Like previous works [9, 45], we support queries that are conjunctions of any number of single-column predicates, while disjunctions can be transformed to conjunctions using the inclusion-exclusion principle. (2) Any single predicate for A_i can be an equality predicate (e.g., $A = a_i$), an open range predicate (e.g., $A \geq l_i$) or a close range predicate (e.g., $l_i \leq A \leq h_i$). Here, we use R_i to denote the range if A_i is a range predicate. For instance, in the above examples, $R_i = [l_i, A_i.\max]$ or $R_i = [l_i, h_i]$. Since our method will transform the equality predicate to range (see Section 3), we also abuse R_i to represent the equality predicate for ease of representation. (3) We also support LIKE for matching the prefix, suffix or substring of string attributes, like ab%, %tion and %tri% respectively. As we also transfer LIKE predicates to ranges, Equation 1 can be written as:

$$sel(\theta) = \sum_{t \in R_1 \times \cdots \times R_m} P(t) \tag{2}$$

2.2 Normalizing Flow-based Model
The joint data distribution is modeled via generative models, where GAN [7], VAE [16], Autoregressive [5] and Normalizing Flow (NF) [2, 34] are typical models. However, GAN and VAE perform well on tasks like image generation, but cannot be applied to the CE problem. The reason is that these models directly generate the objects during inference, but do not output the probability density, so it is intractable for them to estimate the cardinality. Although the autoregressive model [5], a type of generative model, has been applied in CE recently, it still suffers from the large domain size problem, as discussed in Section 1. Therefore, we adopt the Normalizing Flow, another representative generative model to solve the CE problem.

Generally speaking, NF provides a method for modeling flexible probability distributions over *continuous random variables*. It
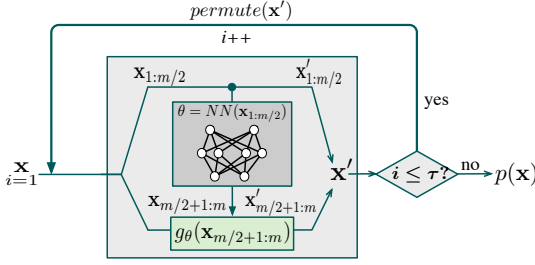
Figure 2: An Example of Coupling-based Flow models.

can transform a complex probability distribution into a simpler distribution (e.g., a standard normal) using a sequence of *invertible* and *differentiable* transformations. These transformations can be parameterized by neural networks. Formally, suppose $\mathbf{x}$ is an m-dimensional dataset that we want to learn a joint distribution. The basic idea of NF is to represent $\mathbf{x}$ as the output of a sequence of transformations (uniformly denoted by $\mathbf{f}$) of a real vector $\mathbf{u}$ sampled from a simpler distribution $\pi(\mathbf{u})$, i.e., $\mathbf{x} = \mathbf{f}(\mathbf{u})$ where $\mathbf{u} \sim \pi(\mathbf{u})$ [30].

Leveraging the transformation of the NF, the probability density of $\mathbf{x}$ can be obtained using a change of variables,

$$p(\mathbf{x}) = \pi(\mathbf{f}^{-1}(\mathbf{x})) \left| \det\left(\frac{\partial \mathbf{f}^{-1}}{\partial \mathbf{x}}\right) \right|. \qquad (3)$$

For example, given a data point after pre-processing, e.g., $\mathbf{x} = (-1.05, 2.31, 0.27)$, as the input of the NF model. It infers the estimated probability density of this point, e.g., $p(\mathbf{x}) = 3.18$, based on learned data distribution. Then the probability densities of multiple data points can be utilized to compute the cardinality of a query.

Since we need to compute $\mathbf{f}^{-1}$ and its Jacobian matrix in the above equation, $\mathbf{f}$ has to be *invertible* and *differentiable*. Intuitively, the transformation not only maps between $\mathbf{x}$ and $\mathbf{u}$, but also quantifies the change of density by the Jacobian matrix. For efficiency, $\pi(\mathbf{u})$ is usually simple, e.g., standard normal distribution.

In NF, $\mathbf{f}$ should be carefully designed for *invertible*, *differentiable* and *efficient* computation, so we adopt the coupling transformation [2, 28, 50] for $\mathbf{f}$, which consists of a series of coupling layers, denoted as a loop in Fig. 2. The number of layers cp is a hyperparameter, say 5. Each coupling layer has the same input/output dimension, which is designed by the following steps:

- Divide the input $\mathbf{x}$ into two equal parts: $[\mathbf{x}_{1:d}, \mathbf{x}_{d+1:m}]$, where $d = \frac{m}{2}$.
- Feed the former part into a lightweight neural network (e.g., MLP), $\theta =$ MLP$(\mathbf{x}_{1:d})$.
- Set $\mathbf{x}'_{1:d} = \mathbf{x}_{1:d}$ directly.
- Set $\mathbf{x}'_{d+1:m} = g_\theta(\mathbf{x}_{d+1:m})$, where g is a differentiable and invertible element-wise function parametrized by θ. Return $\mathbf{x}' = [\mathbf{x}'_{1:d}, \mathbf{x}'_{d+1:m}]$.
- $\mathbf{x}'$ is permuted and fed into the next coupling layer. Note that different coupling layers have different parameters for capturing correlations of multiple columns.

Hence, $\mathbf{f}$ is *invertible*, i.e., given $\mathbf{x}'$ in each layer, we can simply restore $\mathbf{x}$. The reason is that $\mathbf{x}_{1:d}$ equals to $\mathbf{x}'_{1:d}$, and we can get $\mathbf{x}_{d+1:m}$ from $\mathbf{x}'_{d+1:m}$, $\mathbf{x}_{1:d}$ and the invertible g. $\mathbf{f}$ is naturally *differentiable* because g is differentiable. It is efficient as each coupling layer has lightweight network structures. From the above steps, we can see that the Jacobian matrix J of a coupling layer is lower

triangular, which means that the determinant of J can be computed efficiently in $O(m)$ as the product of the diagonal elements.

For training the NF, given a dataset $D = \{\mathbf{x}^{(i)}\}_{i=1}^{N}$, a flow is trained to maximize the total log likelihood $\sum_i \log p(\mathbf{x}^i)$. The CE problem can be solved by transforming each tuple t to a data point $\mathbf{x}^{(i)}$ and modeling the joint probability distribution.

2.3 Related Work

Query-driven learned CE methods. They take CE as a regression problem. In the training step, they collect a pool of queries with their real cardinalities as labels, extract the query features and encode them as a vector, and then train a model to map a query to its cardinality. For inference, a query is encoded to a feature vector, fed into the regression model and the CE result is derived. Different models are used, including fully connected neural networks [3, 29], convolutional neural networks [18], recurrent neural networks [29, 37]. In general, query-driven CE methods need a large amount of training data, i.e., queries. If the distribution of queries shifts, the model is not likely to behave well. Therefore, query-driven approaches are expensive and not generalizable enough.

Data-driven learned CE methods. Data-driven methods are proposed to learn the joint distribution of data points in the relational table in the training stage. When inference, they leverage the model to infer the probability of tuples satisfying the query predicates. There are mainly two categories of data-driven learned methods.

(1) Sum-Product network [11]. DeepDB uses sum-product networks to learn the joint distribution. The basic idea is to divide the table into clusters of rows and columns recursively. Then it uses sum nodes to combine different row clusters. For column clusters, it assumes that they are independent and utilizes product nodes to combine them. Although it can capture the joint distribution, it is not accurate because the independence assumption is made.

(2) Autoregressive models [9, 44, 45]. The autoregressive model factorizes the joint distribution into conditional distributions using the multiplication principle. However, the methods cannot handle large domain size data well. Specifically, Naru [45] and DQM-D [9] require to compute the embeddings of each data point, so a large domain size column induces a large number of parameters, leading to high training cost and large model size. Although NeuroCard [44] can alleviate this problem by factorizing the column into several sub-columns, it sacrifices accuracy. Thus existing data-driven methods cannot capture dependencies between columns and cannot handle large domain size, and thus FACE is proposed to address this issue.

3 FACE FRAMEWORK

We propose FACE, a cardinality estimation framework using the NF model. In this section, we first introduce the basic idea of using NF (Section 3.1), and then the overall architecture (Fig. 3) of FACE (including training (Section 3.2) and inference (Section 3.3)).

3.1 NF for Cardinality Estimation

We first present the overall framework of FACE, discuss the advantages and summarize the challenges.

Overall Framework. FACE learns a *continuous joint distribution* of the input data using NF. As Fig. 3 shows, it first takes as input the original data. Then for different columns with different data types, FACE uses appropriate encoding strategies and generates the encoded data that can be fed into the NF model (see Section 3.2).
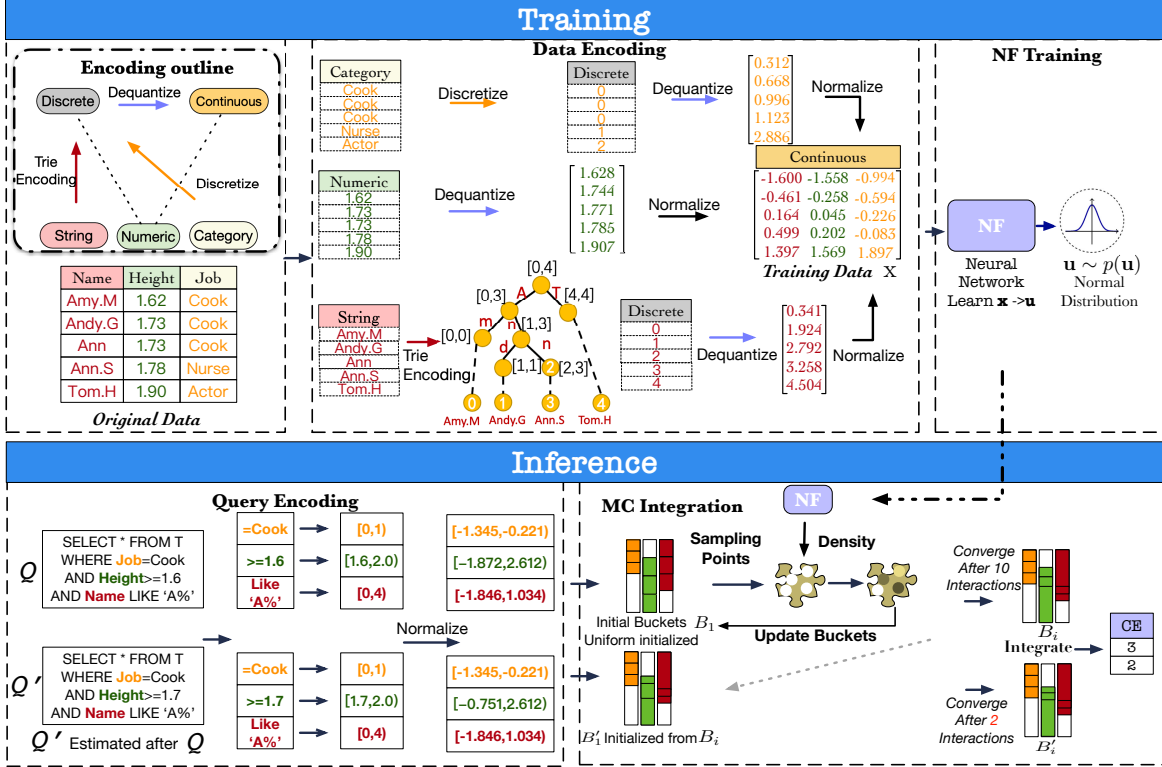
Figure 3: The Framework of FACE.

After training, we can compute the probability density of each data point using the NF model, i.e., $p(\mathbf{x})$.

For inference, as the learned joint distributions are continuous, we use Equation 4 to estimate the cardinality on range predicates:

$$sel(\theta) = \int_{\mathbf{x} \in R_1 \times \cdots \times R_m} p(\mathbf{x}) \, d\mathbf{x}. \tag{4}$$

Note that not all predicates are range predicates. Therefore, to apply Equation 4, we transfer other predicates (including equality, Like predicates) to ranges (see Section 3.3). Afterwards, as the inference part in Fig. 3 shows, we sample some data points from these ranges (see Section 6), call NF model to estimate the probability density of these data points and finally compute the estimated cardinality using Monte Carlo (MC) integration [22].

Advantages. (1) FACE can capture the column dependencies because in each coupling layer as shown in Fig. 2, the former half part of columns interact with the latter half part. Then the output is permuted and the above step is repeated several times, and thus the dependency between columns is likely to be fully captured. (2) NF can naturally support continuous data well, which is a typical type in large domain size data. It takes as input continuous data with simple transformations (e.g., normalization) rather than embedding, which leads to large model size and high training costs.

Challenges. (1) Besides continuous data, there are several common data types (e.g., categorical, discrete and string data) in a relational table, and thus using NF to support them is challenging. To address this, we propose an effective dequantization method to make any type of data continuous (see Section 4) and build an index to tackle Like predicates with string data (see Section 5). (3) The repetitive sampling is time-consuming in the inference step, so an acceleration method is proposed in Section 6.

3.2 Training

The upper part of Fig. 3 outlines the training process of FACE. It first takes as input batches of tuples in T and encodes them in order to make them be well modeled by NF. Then the model is trained using NF with maximum likelihood estimation.

3.2.1 Encoding the Training Data. Generally, there are three common types of data in databases: *numerical*, *categorical* and *string*. Since NF model naturally works on continuous data, we need to conduct a preprocessing step on different types of data. As shown in ***encoding outline*** of Fig. 3, numerical data can be classified into *continuous* data and *discrete* data. The former one can be handled directly by NF, and we propose a dequantization method to make the discrete data continuous. For categorical data, we discretize them as done by most existing works [9, 45], and then tackle them as discrete data. For string data, we encode them using a tree index, and use trie encoding to convert strings to discrete data. Next, we introduce the above steps in detail using the example in Fig. 3.

Categorical data. We transform the categorical data into continuous space. We first convert them into discrete data ($E(a_i) \rightarrow w$). For example, $E(Cook) \rightarrow 0$. However, if we fit discrete data directly with a continuous density model, e.g. NF, it will produce a degenerate solution that places all probability mass on the discrete data points. Therefore, we use the *dequantization* [13, 40] method that adds noise to discrete data over the width of each discrete bin. Then this method makes the data being continuous, and thus *the probability of each discrete point can be converted to integration over a range.* For the Name attribute in the example, the values are encoded to $\{0, 1, 2\}$, and they have the equal length of bins, i.e., $bin = 1$. Then for each discrete point with value $v \in \{0, 1, 2\}$, we add a noise

that follows a certain distribution in $[0, bin]$, say uniform distribution. Then $E(\text{Job})=\{0, 0, 0, 1, 2\}$ may become more continuous like $\{0.312, 0.668, 0.996, 1.123, 2.886\}$, which is fed into NF for training after normalizing. When we want to predict $P(Job = Cook)$, i.e., $P(0)$, hopefully, we can compute it by integration over $[0, 1]$, i.e., $\int_0^1 p(x)dx = 0.6$, where $p(x)$ is learned by NF. The dequantization technique is significant in accuracy improvement for Flow models, so in Section 4, we propose a more effective strategy considering the continuity of noised data.

Numerical data. As discussed above, we encode categorical data to discrete data and then dequantize it. Therefore, for discrete data in numerical data, we can directly dequantize it using the above method. For continuous data, intuitively, we feed it into NF without any processing. However, any data in a computer is represented by a finite number of bits, and continuous data requires an infinite number, so there is no real sense of continuity. To make data more continuous, we also apply dequantization on these seemingly "continuous" data, which makes a probability density much easier for NF to learn. For example, in attribute `Height`, the length of bin is $1.78 - 1.73 = 0.05$, so we add noise in $[0, 0.05]$. Then the two 1.73 become 1.744 and 1.771.

String data. `Like` predicates are widely used for string data in database queries. To handle this, for `Like` predicates with patterns `ab%`, `%tion` and `%tri%`, we build a trie-based index to encode each string to discrete data so that the `Like` predicates can be converted to range predicates. Then we can use the above method to further encode these discrete data using dequantization and feed into NF. Specifically, we initialize a global `ID` as 0, and then traverse the trie in depth first search (DFS) order. For each leaf node (corresponding to a full string), we assign the node with the current `ID`, and increase `ID` by 1. For example, the DFS order of `Name` in Fig. 3 is `Amy.M`→ `Andy.G`→ `Ann`→ `Ann.S`→ `Tom.H`, and they are encoded as $[0, 1, 2, 3, 4]$.

Normalization is applied after all the above transformations to get the final training data, which is sent to the NF model for training.

Flow-model Training. Data encoding transforms each tuple in table T to $\mathbf{x}$ with the same dimension. Then $\mathbf{x}$ is fed into NF model for training iteratively. We also use the same loss (maximum likelihood estimation) as introduced in Section 2.2.

3.3 Inference
Given a model and a query, we show how to utilize the NF model to estimate the cardinality of the query. First, we introduce how to encode queries for inference. Second, considering the query similarities, we illustrate how to accelerate the inference step.

3.3.1 Query Encoding.
In this paper, we do not distinguish between point and range queries, since we convert every equality predicate into a range. The reason is that the equality predicate is applied on categorical and discrete data that are modeled as continuous data by NF. In fact, in our scenario, query encoding is equivalent to encode the predicates of the query, i.e., how to transfer the predicates (including equality and `Like` predicates) to range predicates.

Equality predicates. We first encode the equality predicate $A = a_i$ to a range. If a_i is a categorical value, we encode it to the same discrete value as the encoding in the training phase i.e., $E(a_i) \to w$. Then the range is constructed by $[w, w + bin)$, where bin is the corresponding bin width of w. For example, the predicate `Job` =

Cook is encoded as $[0, 1)$. Then the cardinality can be estimated by integration over the range. If a_i is a discrete value, we can directly construct the range.

Range predicates. For predicates with a close range, we can compute integration straightforwardly over the range. For open ranges, we will simply find the MAX/MIN of the attribute and construct the range. For example, the predicate `Height` $\geq$ `1.6` is encoded as $[1.6, 2.0)$ because 2.0 is the MAX of the `Height` attribute.

`Like` predicates. We also convert `Like` predicates to ranges based on the trie-based index. For a prefix `Like` predicate (e.g., `An%`), we search `An` on the tree, and the node is associated with the range corresponding to `An%`, i.e., $[1, 3]$. For suffix predicates (e.g., `%on`), we search on a suffix-based Trie. For substrings (e.g., `%on%`), we construct multiple ranges based on prefix-based Trie (see Section 5).

All of the above ranges are normalized for online estimation.

3.3.2 Similarity-based CE Acceleration.
Given the trained NF model, we compute the probability density of each data point. Together with the given ranges, ideally, we want to obtain the cardinality by computing the integration over these ranges using Equation 4. Unfortunately, the integration is infeasible to compute, because it has no closed-form solution. Thus, MC integration [31] is applied to approximate this. The basic idea is to *sample a number of data points from the range*, compute the probability density of them using NF and integrate the results to estimate the cardinality. Thus, *sampling* largely determines the efficiency and accuracy of inference.

Adaptive importance sampling. A simple sampling strategy is uniformly sampling from the range R_i, but it degrades the accuracy because data in R_i may not be uniformly distributed. Therefore, we adopt the adaptive importance sampling [22, 31] strategy as shown in Fig. 3. It samples from the range adaptively according to the data distribution, described by buckets for different attributes. At the beginning, we initialize equi-width buckets (B_1 in the example) as we know nothing about the distribution. Then we sample data points from the buckets, use NF to compute the probability density of them, and update the buckets. We repeat the above steps until convergence, and use the buckets (B_i) that can accurately describe the distribution of range data to conduct the MC integration. We can observe that although the method can capture the data distribution, the repetitive sampling leads to inefficiency, so we propose to accelerate this process based on query similarities.

Accelerate subsequent queries. In real scenarios, queries can arrive at any time. For example, in Fig. 3, Q' comes after Q and they seem to be similar. We can measure the similarity of queries by comparing each pair of ranges of two queries. We observe that ranges of similar queries are mostly overlapped, and thus their sampled data follow similar distributions. Therefore, we initialize the buckets of the new arrival query using that of the most similar one (Initialize B_1' using Q). In this way, we can obtain B_i' in much fewer iterations, making the inference more efficient.

In Section 6, we introduce how to compute the query similarity. Based on that, we will also illustrate how to accelerate the inference step using buckets in detail.

3.4 Joins
FACE can also support join queries in two ways: *Single-Model* and *Multi-models*.

Single-Model follows existing solution [11, 44] that leverages one cardinality estimator to learn the distribution of each table and joins of multiple tables in the schema. We first generate a full-join table by using full outer joins to join all tables. and then add some columns to the full table. Note that it is expensive to generate all join tuples, we sample some joined tuples [47] (can be seen as samples of the full-join table). Next, it trains a single model for the full-join table using our method and then uses the estimation method to support both a single table and multiple tables (with join queries). Note that the full-join table may contain duplicated tuples for a query and NULL values. To address this issue, we add additional columns and feed the table with additional columns into our model. After training, given a query, we use the trained model to estimate the cardinality. The difference is that we will further leverage the values in additional columns to correct the probability densities considering the join types, redundant tuples, and NULL values.

For the Single-model, the full-join table may be very sparse and the trained model may not be effective for different queries. To address this, we can train multiple models, i.e., training a model for each possible join query, and then given a query, we use the corresponding model to estimate the cardinality. However, it is rather expensive to enumerate all possible joins and build a model for each join. To alleviate this issue, we can generate all possible join templates based on historical queries (a join template is a join query by removing all predicates and only keeping the join structure), train a model for each template, and then the number of models to be trained can be reduced. To summarize, the advantage is that it provides more fine-grained estimation than the *Single-Model*. However, it needs additional join template information, and may consume larger memory when the number of models is large.

4 DEQUANTIZATION

In this section, we will introduce the spline dequantization designed by us for making data "more continuous", which is inevitable if one wants to encode data for feeding into NF. We first show the basic idea of the dequantization and then how to implement it.

Basic Idea of dequantization. We begin with an example for modeling a continuous distribution of an attribute A_i with 5 categories. If we encode them to discrete data (Section 3.2.1) and use NF to fit them, we will derive a probability density function (PDF) as shown in Fig. 4 (a). This method has two limitations. On the one hand, fitting a continuous density model to discrete data will produce a degraded solution [12] because all the probability mass is placed on discrete data points. On the other hand, while inference, it is infeasible to compute the probability of a category using the PDF p because the integral interval is unknown. Therefore, the dequantization technique has to be applied.

Dequantization distribution. As discussed in Section 3.2.1, dequantization is utilized to add noise on discrete data so that NF can learn the continuous probability distribution more easily. Formally, given a discrete data point x, the noise u can be generated following a *dequantizing distribution* $q(u|x)$, $u \in [0, bin)$. Here bin is the width of the discrete bin of x, which is the difference between x and the smallest value bigger than x in A_i. After dequantizing all values that equal to x, these values will all lie in the corresponding bin $[x, x + bin)$, so the integration over the bin precisely captures the probability of x.
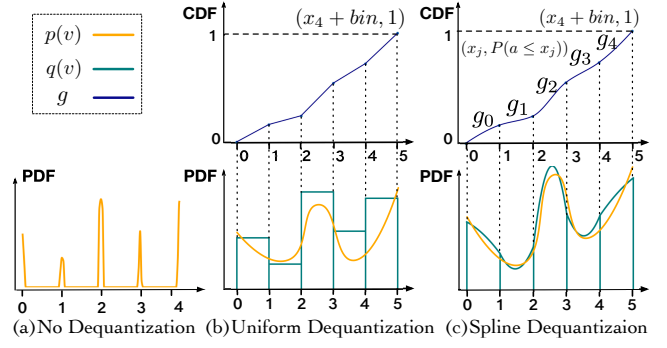

Figure 4: Visualization of Dequantization Methods.

Then the noise is generated based on distribution q, and each discrete value becomes $v = x + u$, so v is the dequantized data (Note that for explicit representation, we use v to denote data after dequantization, while in other Sections, x is still used to denote the data after all pre-processings). Recap from Section 3 that NF learns the PDF p based on these dequantized data. Then the probability of any discrete point, $P(x)$, can be computed by integration. Ideally, we hope that $P(x) = \int_{x}^{x+bin} p(v)dv$, but in fact it cannot hold exactly in real case, which can be well approximated by a sophisticated dequantization distribution.

Motivation of spline dequantization. There exist many optional dequantization distributions, and uniform dequantization [40] is a representative one. Suppose that we use it to model $q(v|x)$, which generates noise uniformly for each discrete point. In our example, these data points have $bin = 1$. Fig. 4 (b) visualizes the distribution (green rectangles) of dequantized data, i.e., $q(v) = \mathbb{E}_{x \sim P}[q(v|x)]$. *The objective of a well-performed dequantization method is to make p learned by NF well fit the data dequantized by q.* However, it is hard for NF to fit the data dequantized by uniform dequantization. The reason is that p is a continuous distribution that we want to learn, but it is naturally difficult to learn from data obtained by a discontinuous distribution q. Also, other existing works [12, 13] cannot guarantee the continuity property.

Therefore, we propose a spline dequantization technique that *utilizes spline interpolation to construct a continuous dequantizing distribution for each attribute.*

Implementation of Spline Dequantization. Next we discuss how to dequantize discrete data using the continuous spline dequantization distribution. The general solution consists of two steps. (1) Construct a cumulative distribution function (CDF) of each attribute using spline interpolation. (2) Use the CDF to generate dequantized data v, which will be leveraged by NF for training. The basic idea of the above steps is that, to derive a continuous dequantization distribution q, we construct a *continuously differentiable* CDF. Hence, since q is the derivation of the CDF, q is naturally continuous.

For example, as shown in Fig. 4 (b), the CDF of the uniform dequantization is not continuously differentiable, so q is not continuous and the generated dequantized data is hard to fit. Therefore, it requires to construct a high-quality CDF.

CDF construction. Considering a discrete attribute A_i with domain size $s = |A_i|$, we abuse a to denote the random variable that A_i can take. For each $x_j \in A_i$ (x_j denotes the j-th smallest value in A_i), we can easily compute the probability that the attribute

will take a value less than x_j, i.e., $P(a < x_j)$, which can be used to construct a CDF. To be specific, first, we plot the points, i.e., $(x_1 = 0, P(a < x_1))$, $(x_2 = 1, P(a < x_2))$, ..., $(x_j, P(a < x_j))$,..., $(x_s + bin, 1)$ on coordinates, as shown in Fig. 4 (c). Second, we use Monotone Piecewise Cubic Spline Interpolation [4] to compute a piecewise polynomial function, namely the CDF (denoted by g). It consists of s polynomial pieces, each of which (g_j) is a cubic function corresponding to values in range $[x_j, x_{j+1}]$. The reasons why we use such a method to construct a CDF are three-fold. (1) The spline interpolation holds *monotonicity*, which is necessary to represent the naturally monotonic CDF. (2) The spline interpolation guarantees the continuously differentiable property, so q is continuous because it is the derivative of the CDF. As shown in Fig. 4 (c), NF can well fit the dequantized data generated from such dequantization distribution. (3) The computation of spline interpolation is efficient.

Generate dequantized data. Next we will generate dequantized data using the CDF, which comprises two phases. Suppose that we want to dequantize a discrete value x_j. First, we sample a probability pr from the range $[g_j(x_j), g_j(x_{j+1})]$. Second, we compute the inverse function g_j^{-1}, which maps each probability between $g_j(x_j)$ and $g_j(x_{j+1})$ to a value between x_j and x_{j+1}. g_j^{-1} can be calculated fast and easily, because g_j is a cubic function. Then we obtain dequantized $v = g_j^{-1}(pr)$.

Remark. One may wonder why we do not use q to infer the cardinality directly rather than the PDF p. The reason is that q is the marginal distribution of each attribute in our example, but what we want to learn (the PDF p) is a joint distribution. To address this issue, we can extend spline dequantization to multiple attributes by constructing continuously differentiable CDF on multi-dimensions [1, 8]. As it is prohibitively expensive to construct q on all dimensions, we usually use small dimensions (1 or 2 dimensions).

5 STRING ENCODING AND INFERENCE

To support Like predicates in data-driven CE that suffers from challenges of large domain size and inefficient inference, we build a trie-based tree to index strings, encode each string to discrete data based on the trie and convert Like predicates to range predicates.

5.1 Trie Encoding

We first build a trie-based index and introduce how to encode strings based on it. Given $A_i = \{art, ate, car, cat\}$, we can build a trie $\mathcal{T}$ as shown in Fig.5 (a). The leaf nodes (*green*) denote strings in A_i, and the non-leaf nodes (*yellow*) represent the prefixes[1].

Trie encoding. We aim to encode the strings in A_i, i.e., these leaf nodes. Each node n in $\mathcal{T}$ records three kinds of information. (1) The string ($n.str$) represented by the node. (2) Encoding ID ($n.e$) of a leaf node, which is a unique ID of the node. We can assign each leaf node an ID in DFS order. Note that non-leaf nodes do not need encodings. (3) The encoding range ($n.r$) denotes the range ($n.min, n.max$) of encodings among strings in the subtree rooted at n, i.e., $n.min(n.max)$ is respectively the minimal (maximal) ID of leaf nodes under n. Now strings in A_i are encoded as discrete values. After dequantizing, they can be fed into NF for training in

---

[1]If there are some strings in A_i that correspond to non-leaf nodes in the trie, we can easily add dummy leaf nodes to represent them.

(a) Trie Encoding Example

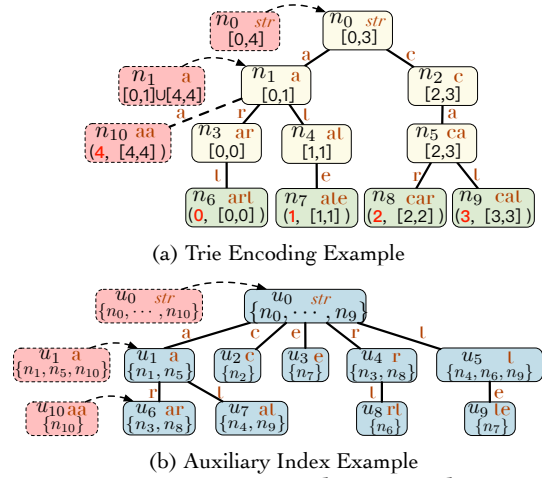(b) Auxiliary Index Example

Figure 5: String Encoding Example.

the same way as aforementioned numerical data. Next, we discuss how to conduct inference.

Inference of prefix-based predicates. For prefix-based predicates, i.e., str%, if there exists a node with $n.str$ =str, we will integrate the learned p over the range $n.r$. For example, suppose that a predicate is A_i Like c%. On the Trie, we match c with $n_2.str$, fetch the range ([2,3]) and estimate the cardinality.

Inference of suffix-based predicates. For suffix-based predicates, i.e., %te, we also tackle them using trie as follows. For each string attribute A_i, we add another column A_i', where each string value is the one-to-one reverse of that in A_i. In the above example, $A_i' = \{tra, eta, rac, tac\}$. Then similar to prefix-based predicates, we use A_i' to build another trie for training and inference.

Another Like pattern is ***substring***, i.e., %str%. It is more challenging to estimate because we cannot directly locate which strings contain str using the trie. Next, we discuss how to solve this case.

5.2 Inference of Substring Predicates

We discuss how to find qualified strings satisfying the substring predicates and transform them to several ranges for efficient inference. For example, for a predicate Like %at%, there are two nodes (ranges) that should be considered in the inference step. To this end, we build an auxiliary Trie $\mathcal{T}_a$ to index the nodes in $\mathcal{T}$, i.e., pre-computing some nodes that have common strings. We first introduce how to build $\mathcal{T}_a$, and then use it to support inference.

Auxiliary index $\mathcal{T}_a$. $\mathcal{T}_a$ tries to match all possible substrings with nodes in $\mathcal{T}$. Hence, given a substring, we efficiently find the matching nodes as well as ranges, and CE is computed by their integrations. Assuming that the character set size is C and the maximum length of strings is M, theoretically, the number of possible substrings is $O(C^M)$, which is prohibitively expensive to enumerate. To address this, we build trie $\mathcal{T}_a$ layer by layer to prune the space.

Specifically, each node in $\mathcal{T}_a$ maintains two types of information. One is the substring, denoted by $u.str$. The other one is a set $u.s$ of nodes in $\mathcal{T}$, s.t., $\forall n \in u.s, n.str$ has the pattern %u.str. For example, $u_7.str =$ at, and thus $u_7.s = \{n_4, n_9\}$ because $n_4.str =$ at and $n_9.str =$ cat. To build $\mathcal{T}_a$, we start with the root that $u_0.str =$ NULL. Then for the second layer, we expand the root by generating C children, each of which corresponds to a character. Then we fill the $u.s$ in the second layer by searching on $\mathcal{T}$. Next, we repeat the

above steps iteratively. To make this efficient, we propose a simple yet effective pruning strategy. We limit the height of the tree to H, say $H = 3$. In this way, the space and time complexity of the search can be greatly reduced, but for inference, one has to explore $\mathcal{T}$ if just the prefix of str matches a leaf node in $\mathcal{T}_a$ (see **Case 2**). Fig. 5 (b) shows the example with $H = 3$.

Inference for substrings. Given $\mathcal{T}_a$, $\mathcal{T}$, and a substring predicate, we introduce how to estimate the cardinality for three cases of str.

Case 1: $\exists u \in \mathcal{T}_a, u.str=$ str. Then $\forall n \in u.s$, we union all ranges, i.e., $n.r$ and integrate over them using the NF model. For example, suppose that we have a predicate Like %a%. Since in $\mathcal{T}_a$, $u_1.str =$a, and $u_1.s = \{n_1, n_5\}$, we can then integrate over $[0, 1] \cup [2, 3]$, i.e., the union of $n_1.r$ and $n_5.r$.

Case 2: $\forall u \in \mathcal{T}_a$, $u.str \neq$str, but $\exists u \in \mathcal{T}_a$, $u.str$ is the prefix of str and u is a leaf node of $\mathcal{T}_a$. In this case, $\forall n \in u.s$, we check the descendants of n in $\mathcal{T}$, and if there exist nodes that contain str, their ranges will be used for estimation. Suppose a Like %art% predicate. In $\mathcal{T}_a$, we go to u_6 and it is a leaf. Then we iterate descendants of nodes in $u_6.s$, i.e., n_3 and n_8 in $\mathcal{T}$, and find that $n_6.str=$art. Hence, we return $n_6.r = [0, 0]$ for estimation. Note that $[0, 0]$ is a discrete point, we address this using the method as discussed in Section 3.2.1.

Case 3: If str does not satisfy the above two cases, we come to the last one, which indicates that there is no string in A_i satisfying the predicate. Given a Like %act% predicate, after coming to u_1, there is no edge c, indicating that act does not exist in A_i.

Complexity analysis. In the last layer of $\mathcal{T}_a$, the number of nodes is at most C^H, and $|u.s|$ of each leaf node is $\frac{|\mathcal{T}|}{C^H}$ on average, where $|\mathcal{T}|$ denotes the number of nodes in $\mathcal{T}$. Thus, the complexity is $O(\frac{|\mathcal{T}|}{C^H})$ because for $\forall n \in u.s$, it takes constant time to search on $\mathcal{T}$.

Discussion of string updates. Our data structure supports data updates by efficient incremental training. (1) *Insert.* For inserted string str', if it can be found in $\mathcal{T}$, we do not change anything. Otherwise, we insert it on $\mathcal{T}$, assign a new encoding and update the range of its ancestors. For example, suppose that str'=aa. We insert a node n' and encode it as $n'.e = 4$,$n'.r = [4, 4]$. Then its ancestors combine with $n'.r$ (the dotted red nodes in Fig. 5 (b)). $\mathcal{T}_a$ also changes. If many strings are inserted, a training from scratch is triggered. (2) *Delete.* Deletion does not have a large impact on training. But for inference, similar to *insert*, we need to delete the node and update the ranges of its ancestors and $\mathcal{T}_a$.

6 INFERENCE ACCELERATION

In this paper, we propose to use adaptive importance sampling (AIS) [22, 31] to conduct the inference. We first introduce its motivation and the basic solution in Section 6.1. Since AIS is time-consuming and we observe that similar queries can be accelerated through sharing sampled data, we discuss how to leverage this property to make the inference more efficient (Section 6.2).

6.1 Adaptive Importance Sampling

Basic Idea. For inference, as discussed in Section 3.3, given the trained model p and predicates of a query Q, we need to first convert the predicates to ranges and integrate over them (Equation 4). However, as shown in Fig. 6, the probability density function p is always too complicated to integrate, so MC integration [22, 31] is always applied to approximate the result.
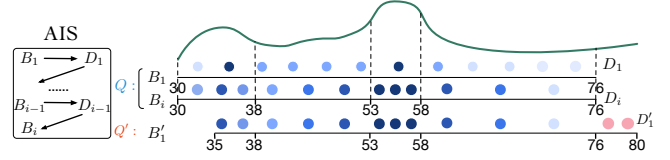

Figure 6: An Example of Adaptive Importance Sampling.

Naive solution. The basic idea is to sample K data points uniformly for each range R_i (corresponding to each attribute), concatenate them to K tuples, compute their probability densities, and use them to get the integration. However, as shown in Fig. 6 (the first line, B_1), this sampling method fails to generate enough data points (dark points in the Figure) in high-probability-density areas, leading to an inaccurate approximation.

AIS. AIS [22, 31] is proposed to split each range R_i[2] into a sequence of successive buckets $B = [b_1, b_2, ..., b_{|B|}]$, and then sample uniformly in each bucket, in order to make sampling points following the distribution of p as exactly as possible. The bucket number $|B|$ (e.g., 10) and the ranges are given, and the AIS task is to adjust the length of each bucket. Intuitively, the shorter a bucket is, the higher the probability densities of the corresponding points in the bucket are, i.e., *more important*. We adjust the length of each bucket *adaptively* until converge, i.e., adjacent buckets differ a little.

In the first iteration, AIS initializes a bucket sequence B_1, where each bucket has the same length, and then samples $\frac{K}{|B|}$ points (denoted by a set D_1) uniformly in each bucket (this step is equivalent to the naive solution). Next, based on D_1, AIS computes a new sequence B_2 with the objective that $\forall b \in B_2$, they have the same total probability, which is computed by data points of D_1 lying in each b. Then we use B_2 to sample D_2 using the same sampling strategy. We repeat this until B_{i+1} (generated from D_i) differs a little with B_i, i.e., convergence. As Fig. 6 shows, using AIS, more data are sampled in high-density areas. Finally, we use $D_1, D_1, ...D_i$ to compute a weighted MC integration [22, 31], where D_i will have a large weight because we think that points in D_i are sampled mainly based on p.

Observation. AIS is time-consuming because it always needs multiple sampling iterations to converge. However, we observe that similar queries always generate similar samples, so we can share these samples to reduce the number of samples, so that similar queries take less time to converge. Next, we first define how to measure the query similarity and then show how to do acceleration.

6.2 Sampled Data Sharing

Query similarity. Supposing the table has m attributes, we construct m ranges for a query Q, i.e., $R_1, R_2, ..., R_m$, and each range is denoted by $R_i = [l_i, r_i]$ (see Section 2.1). Given another query Q', we define $sim(Q, Q') = \frac{1}{m} \sum_{i=1}^{m} \frac{|R_i \cap R'_i|}{|R_i \cup R'_i|}$. The similarity score is between $[0, 1]$. The higher $sim(Q, Q')$ is, the more similar Q and Q' are. For substring-based Like predicate, we use the union of ranges to compute the similarity.

Share with subsequent queries. In many cases, queries come in the form of streaming data. If the CE of each query requires to

---

[2]Substring-based predicates are likely to generate multiple ranges. Our solution can sample them simultaneously.

Table 1: Real Datasets.

Dataset	Size (MB)	Rows	Cols/Cate	Dom	Joint
Power	95	2.05M	6/0	$\approx$2M	10^{37}
IMDB	123	4.74M	6/5	$[2, 1M]$	10^{16}
BJAQ	15	380K	5/0	$[1K, 2K]$	10^{15}

sample iteratively for many times until convergence, the performance of the system will be greatly reduced. Fortunately, a query can leverage the information of the previous most similar query to accelerate the convergence. To be specific, given a new coming query (e.g., Q' in Fig. 6), we find the most similar query among all queries that have been estimated, say Q. Then we share D_i to initialize the first bucket sequence of Q', i.e., B'_1, because the ranges in both queries have similar distributions. However, as shown in Fig. 6 (c), their corresponding ranges, i.e., R_i and R'_i are a little different, so we have to slightly adjust D_i to fix the difference. Specifically, on the one hand, if there exist data points of D_i that do not lie in range R'_i (e.g., range [30,35] in Fig. 6), we directly drop them from D_i. On the other hand, if $\overline{R} = R_i \cup R'_i - R_i$ is not NULL (e.g., range [76,80] in Fig. 6), we sample $\frac{K}{|B|}$ points from $\overline{R}$ and add them to D_i. The reason is that these added points do not appear in origin D_i, but they are required in R'_i.

In this way, the number of iterations of Q' can be reduced, and thus the inference will be accelerated. This method will improve the efficiency without sacrificing the accuracy because after the initialization, the following sampling iterations of Q' can still navigate the bucket sequence to further approximate the true distribution p. Note that we cannot store all the sampled data points of all previous queries in reality due to the storage overhead. To address this, we set a storage limit, count the number of times that each query is shared and maintain a priority queue of queries. We discard queries that are not commonly shared when the storage limit is achieved.

7 EXPERIMENT

We have conducted extensive experiments to show the superiority of our proposed FACE framework. We first introduced the experimental settings, and the overall performance of FACE comparing with existing works in Sections 7.1 - 7.5. Then we evaluated our proposed techniques in Sections 7.6 - 7.7.

7.1 Experimental Settings

Dataset. We used three real-world datasets which were widely adopted by existing works [3, 9, 20], and TPC-H, a widely used benchmark. Table 1 showed the information of the datasets. The Cols/Cate meant that the overall number of columns/the number of categorical columns. Dom denoted per-column domain size. Joint referred to the number of entries in the exact joint distribution.

Our datasets covered different properties of data, including different sizes, data types, domain sizes, etc. (1)Power [14] is a household electric power consumption data gathered in 47 months. It has large domain sizes in all columns (each $\approx$ 2M) and all columns are numerical data. (2)IMDB [20] is a movie dataset that originally consists of 21 tables. We selected three tables, Company_name, Movie_companies, Title and joined them to evaluate. Since the join result was too large, we sampled [47] 4,740,297 tuples from the final result following the original distribution. The domain size of IMDB varies a lot, from 2 to 1M. (3)BJAQ [36] includes hourly air pollutants data from 12 air-quality monitoring sites, which has medium domain

sizes (1K-2K). (4)TPC-H is a commonly used synthetic benchmark dataset, which contains 22 query templates. Specifically, we used the query templates to generate 2000 different queries, and reported the average estimation results of these queries.

Baselines. We compared FACE with a variety of typical CE algorithms, including:
(1) PG [33]: Postgres, using histograms with the independence assumption.
(2) Sample [21, 47]: the method sampled a number of records to do CE. The sampled size was set to 1% of each dataset.
(3) MHIST [32]: the method stored all entries in the PDF using a compression technique.
(4) KDE [10, 15]: used kernel density estimation for CE.
(5) lw-nn [3]: a query-driven method that trained a neural network to estimate the cardinality.
(6) lw-xgb [3]: a query-driven method that trained a gradient boost tree to estimate the cardinality.
(7) MSCN [17]: a query-driven method that used multi-set convolutional network.
(8) DeepDB [11]: the method used sum-product network.
(9) Naru [45]: the method used the Autoregressive model.
(10) NeuroCard [44]: the method also used the Autoregressive model, but it could handle large domain size data by splitting columns. It also extended Naru to support multi-table.

We obtained codes of baselines from the authors and an experimental work [43]. For hyper-parameters, we set to default values.

Workloads for testing. For each dataset except TPC-H, we generated 2000 queries for testing in a similar way as [45]. Multidimensional queries containing both range and equality predicates were generated using the following steps: (1) We randomly selected the number of predicates f in a reasonable interval considering the number of columns in the dataset, e.g. [3, 6] for Power. (2) We randomly selected f distinct columns to place the predicates. For numerical columns, the predicate was drawn uniformly from $\{=, \leq, \geq\}$. For categorical columns, we only generated equality predicates, because range predicates on categorical attributes were not practical. We only generated Like predicates on IMDB in Section 7.4. (3) We randomly selected a tuple from the table, and used the attributes of the tuple as the literals. For TPC-H, we used the TPC-H benchmark query templates to generate 2000 queries.

Hyper-parameter Setting. For Power, IMDB, BJAQ, TPC-H, we set the number of coupling layers as $\tau = 6, 6, 6, 5$. In each coupling layer, the MLP consisted of two hidden layers with 108, 108, 56, 48 hidden units respectively. We set the number of buckets $|B| = 100$ and adaptively sample until converge.

Evaluation Metrics. We evaluated different methods from three perspectives: accuracy, latency and model size. For accuracy, we adopted the Q-error metric [18]. It was defined as $Q - error = \max\{\frac{car(\theta)}{\widehat{car(\theta)}}, \frac{\widehat{car(\theta)}}{car(\theta)}\}$, where $\widehat{car(\theta)}$ was the estimated cardinality. We reported the whole q-error distribution (50% (Median), 95%, 99% and 100% (Max) quantile) of each workload. For latency, we reported the average query latency. We also reported model size.

Environment. All experiments were in Python, performed on a server with 32-core CPU, a Nvidia 2080ti GPU, and 128GB RAM.

Table 2: Q-errors, Latency (ms) and Model Size (MB) on 4 Datasets.

(a) Power

Estimator	50th	95th	99th	Max	Latency	Model Size
PG	1.38	15.6	118	$3e^5$	1.25	**0.92**
Sample	1.04	1.97	150	722	2.07	-
MHIST	5.10	135	383	$2e^5$	2070	11
KDE	1.36	18.2	119	1599	**0.33**	-
lw-nn	1.07	4.70	26.8	455	0.59	4.7
lw-xgb	1.04	3.28	8.10	501	0.35	0.94
MSCN	1.13	17.1	176	488	0.76	4.3
DeepDB	1.06	1.91	5.33	537	16.35	3.2
Naru	-	-	-	-	-	-
NeuroCard	1.03	1.51	5.09	158	71	9.9
FACE	**1.02**	**1.16**	**1.60**	**3.00**	10.74	1.2

(b) IMDB

Estimator	50th	95th	99th	Max	Latency	Model Size
PG	2.92	47.3	2768	$1e^4$	**0.15**	**0.16**
Sample	1.03	1.38	5.00	260	1.06	-
MHIST	1.20	3.36	10.4	386	902	9.8
KDE	1.57	9.45	842	1084	0.32	-
lw-nn	1.23	8.89	35.0	405	0.63	4.7
lw-xgb	1.16	11.1	36.8	1198	0.3	0.99
MSCN	1.18	5.04	64.0	2189	0.85	4.2
DeepDB	1.08	1.89	3.39	62.2	1.68	2.64
Naru	-	-	-	-	-	-
NeuroCard	1.02	1.51	2.76	14.9	64	6.2
FACE	**1.02**	**1.21**	**1.54**	**2.85**	11.6	1.2

(c) BJAQ

Estimator	50th	95th	99th	Max	Latency	Model Size
PG	1.46	9.94	30.4	1502	**0.37**	**0.12**
Sample	1.04	1.33	2.51	271	1.06	-
MHIST	1.89	27	209	579	480	8.5
KDE	1.04	1.69	3.91	219	0.51	-
lw-nn	1.12	5.46	14.1	77.4	0.67	1.5
lw-xgb	1.06	4.38	18.2	106	0.39	1.9
MSCN	1.17	2.39	10.5	164	1.03	1.4
DeepDB	1.06	1.91	5.33	472	4.59	0.53
Naru	1.03	1.26	1.54	8.00	12.4	9.2
NeuroCard	-	-	-	-	-	-
FACE	**1.03**	**1.16**	**1.30**	**2.55**	11.8	0.37

(d) TPC-H

Estimator	50th	95th	99th	Max	Latency	Model Size
PG	1.29	21.2	161	489	**0.26**	**0.01**
Sample	1.04	82	228	524	1.67	-
MHIST	1.03	1.18	1.45	6.60	365	5.6
KDE	1.04	2.79	5.44	43.9	0.48	-
lw-nn	1.09	1.55	2.25	11.0	0.85	0.52
lw-xgb	1.05	1.51	2.01	13.5	0.42	0.81
MSCN	1.05	3.72	10.7	445	0.93	0.25
DeepDB	1.04	1.16	1.33	5.00	6.35	0.45
Naru	1.04	1.22	1.39	5.00	8.89	9.80
NeuroCard	-	-	-	-	-	-
FACE	**1.03**	**1.15**	**1.29**	**1.56**	7.90	0.22

7.2 Overall Evaluation

7.2.1 Comparison of Accuracy. Table 2 showed the Q-errors of different CE algorithms. Methods are grouped as traditional, query-driven, and data-driven. The results could be ranked as FACE > Naru/NeuroCard > DeepDB >> lw-nn/lw-xgb/MSCN/Sample > KDE > MHIST/PG in summary. Next, we explained the results.

Generally speaking, the accuracy of FACE was very high on all datasets with different characteristics. We could see from the table that FACE outperformed all the baseline methods on the entire distribution of Q-error for all datasets. For example, the medians (1.02 or 1.03) on these datasets were close to the optimum. Especially, FACE also performed well on errors at the tail (99th, Max). For example, at the Max-quantile in Power dataset, FACE outperformed the second best solution by 50×. As a consensus [45], errors at the tail should be taken more attention because they represent the worst performance of estimators. Unfortunately, they are harder to optimize than the median, and indicate the stability of estimators. Therefore, the results further demonstrated that our solution was a well-performed yet stable estimator. This is because our framework could well model the joint distribution of data with different types.

FACE performed better than Naru and NeuroCard. Since Power and IMDB are datasets with large domain size, it was intractable for Naru to train. Hence, we just reported the results of NeuroCard, which alleviated this problem by factorizing columns. For BJAQ and TPC-H with median domain size, we reported the results of Naru because NeuroCard used the same method. On Power and IMDB, FACE outperformed NeuroCard by more than 50× and 5× at the Max-quantile respectively, because FACE used NF to model

the joint distribution, which was adequate for large domain size data. Although NeuroCard could handle large domain size data, the accuracy decreased because of the column factorization. For BJAQ and TPC-H, we observed that our method still outperformed Naru by 3×. The reason was that our dequantization technique could make our method support data without a large domain size well.

FACE outperformed DeepDB in accuracy by 1-2 orders of magnitude. For example, on IMDB, at the Max-quantile, FACE was 2.85 while that of DeepDB was 62.2, because DeepDB failed to capture the correlations between all columns. FACE performed well because it can address this problem through coupling layers in NF model.

FACE also outperformed these query-driven methods a lot. For example, on Power at the 99%-quantile, FACE had a Q-error of 1.60, but lw-nn, lw-xgb, MSCN were 26.8, 8.10 and 176 respectively. The reason was that query-driven methods relied on the consistence of training and test workload, which was not generalizable enough. For other baselines, FACE outperformed them by 1-3 orders of magnitude because PG assumes independence between columns. Sample could not handle errors at the tail because of 0-tuple problem [37]. MHIST loses information because of the compression and KDE cannot handle multi-dimensional data well by kernel functions.

7.2.2 Comparison of Latency. We also reported the average latency on 2000 testing queries in Table 2. We could see that the latency of FACE (around 10ms on 4 datasets) was applicable in practice. FACE was faster than Naru/NeuroCard, especially on large domain sizes. For example, on Power, FACE was 7 × faster than NeuroCard. The reason was that Naru had to compute all the probabilities of

qualified entries in each domain. Also, the autoregressive model had to be triggered multiple times for computing the conditional probabilities. FACE was fast because it used the data sharing technique to conduct acceleration. Moreover, DeepDB was faster than FACE on most datasets because it does not use deep neural networks to model the data distribution. That was the reason why it could not completely capture the complex correlations between columns. The query-driven methods had higher efficiency because they did not need to sample from range predicates, but purely conducted inference through queries, which was also the reason why the accuracy was low. Most traditional methods were naturally fast because they were very simple, but suffered from low accuracy.

7.2.3 Comparison of Model Size. In this part, we compared the model size with baselines. In fact, the size of each model could be adjusted by changing the network architecture or hyper-parameters. We obtained the model size from the default settings of each baseline or the experimental work [43]. As shown in Table 2, PG had the smallest model size because PG was just related to the number of attributes. Among learning-based methods, FACE almost performed one of the best. The query-driven methods and DeepDB also had a relatively small model size because the former ones did not need to model the complicated data distribution, and the latter one used a lightweight model. For FACE, although the coupling layer used in our model incorporates neural networks, it was still lightweight because of the compact architecture. While for Naru, large domain size led to prohibitively large model size due to the large number of parameters. Therefore, to summarize, FACE used the NF model with high representation ability yet compact size.

7.3 Synthetic Dataset Evaluation

In this section, we evaluated how the accuracy of our models would be affected by two important factors, i.e., domain size and column correlation. To this end, two synthetic datasets were generated in the same way as [43] corresponding to these two factors. Each dataset contained 1 million rows and two columns. The testing queries were generated based on the same method as Section 7.1.

7.3.1 Evaluation of Domain Size. We varied the domain size on the synthetic dataset from 10 to 100,000 on both columns and compared the Q-errors with Naru[3] and DeepDB, two representative data-driven methods. The results in Fig. 7 (a) showed that the Max-quantile of FACE increased from about 1 to 100 along with the domain size increasing. However, the Q-error of DeepDB and Naru have achieved nearly 10^4 and 10^5 respectively. That was, FACE outperformed them by 2-3 orders of magnitude. This indicated that domain size had a much smaller impact on FACE compared with Naru and DeepDB, and more importantly, we could perform well on large domain size data because of the NF model.

7.3.2 Evaluation of Correlations. We varied the correlation between two columns on the synthetic dataset from 0 to 1. When the correlation approached 1, it meant that the columns had strong correlation (dependence), while 0 meant that they were independent. We could see from Fig. 7 (b) that FACE performed the best and was not sensitive to the column correlation. The reason was that the coupling layer in the NF model captured the column correlations. For Naru, the correlation also had little impact on it because the
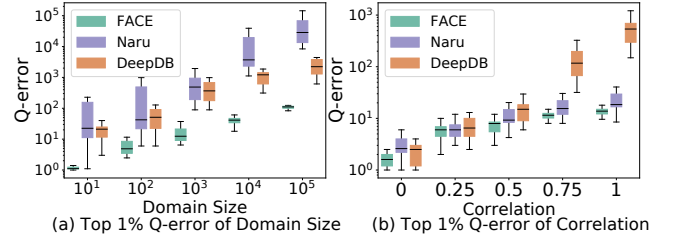
---

[3]When the domain size was large, we applied NeuroCard by factorizing the column.

(a) Top 1% Q-error of Domain Size (b) Top 1% Q-error of Correlation

Figure 7: Evaluation of Synthetic Datasets.

Table 3: Evaluation of Like Predicates.

Estimator	50th	95th	99th	Max	Latency	Model Size
PG	2.31	35.3	207	3118	**2.5ms**	**0.13MB**
E2E	1.51	12.1	54.8	242	5.4ms	43.7MB
FACE	**1.20**	**4.21**	**10.5**	**21.4**	45ms	67.8MB

autoregressive model could capture the dependency. DeepDB performed the worst because it had the independence assumption, so when the correlation became 1, the Q-error of DeepDB was 10^3.

7.4 Like Predicates Evaluation

We evaluated the queries with Like predicates generated in IMDB. Similar to Section 7.1, we first randomly selected f columns, among which we set that at least one column must be a string attribute. Then we randomly selected the pattern among prefix, suffix and substring. Next, a string str should be generated. Specifically, we can sample strings from queries in the benchmark (JOB). However, the number of queries was limited, so we also generated some n-grams with different lengths in the attribute and sampled from them. Totally, we also generated 2000 queries with Like predicates.

We compared with two baselines that supported Like predicates, where E2E [37] was a query-driven cost estimator. We could see from Table 3 that for accuracy, FACE outperformed E2E by one order of magnitude, because E2E was not as generalizable as data-driven methods. FACE outperformed PG by 2 orders of magnitude, because PG cannot capture column correlations.

For model size, we could see that PG only used some simple statistics and thus consumed only $0.13MB$ storage. E2E and FACE had competitive storage usage, because E2E had to store a large number of string embeddings and FACE needed to maintain the trie and auxiliary index structure. For latency, PG and E2E were faster, because the former used the simple statistical technique and the latter used the query-driven method that directly estimated the cardinality using the encoding of queries. FACE was relatively slower, because the data-driven methods needed to sample the data points to estimate the cardinality and searching on the trie index also incurred some overheads.

7.5 Multi-Table Evaluation

In this section, we evaluated the cardinality estimation methods on multiple tables using the widely-used benchmark JOB-light [17]. The results of different methods were shown in Table 4 (results of the baselines are quoted from [44]). The *Single-Model* and *Multi-Models* methods that have been introduced in Section 3.4 were evaluated here respectively, and were represented as FACE-Single and FACE-Multiple.

Table 4: Q-errors and Model Size on `JOB-light`.

JOB-light	50th	95th	99th	Max	Model Size
PG	7.97	797	$3e^3$	$3e^3$	70KB
MSCN	3.01	136	$1e^3$	$1e^3$	2.7MB
E2E	3.51	139	244	272	-
DeepDB	1.32	4.90	33.7	72	3.7MB
NeuroCard	1.57	5.91	8.48	8.51	3.8MB
FACE-Single	1.22	4.84	8.03	8.25	10.5MB
FACE-Multiple	**1.15**	**4.49**	**7.41**	**7.83**	**1.73MB**

As shown in Table 4, we could see that FACE-Multiple performed the best on accuracy because our model captured the joint distribution of different join templates well, which was more fine-grained. We could also observe that for FACE-Multiple, even if we trained a model for each join template, the model size was smaller than the baselines. The reasons were two-fold. (1) Our model size was small because of the compact architecture. (2) Training a model for each join template avoided adding many additional columns to support joins, which might lead to a large model size and higher latency. With a smaller model size and without additional columns, FACE-Multiple achieved an average latency of 11 ms.

Besides, FACE-Single achieved better performance than DeepDB and NeuroCard, because the dataset had some attributes with large domain sizes. But the added columns contained many discrete values, which limited the superiority of the NF model. Using similar methods to learn full outer join distribution resulted in similar model sizes for different data-driven methods. However, the additional columns resulted in higher latency. On average, FACE-Single estimated each query using 60 ms. Comparing FACE-Single and FACE-Multiple, we could see that FACE-Multiple outperformed the FACE-Single and other baselines because it provided more fine-grained models and did not need additional columns with discrete values.
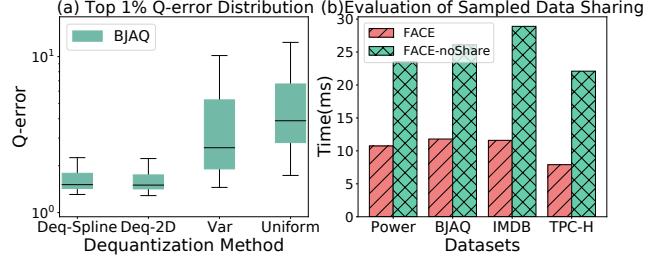
7.6 Variance Evaluation

In this section, we evaluated our proposed techniques including dequantization and sampled data sharing.

7.6.1 Dequantization. We compared our spline dequantization (Deq-Spline, proposed in Section 4) with three baselines: uniform dequantization [42], variational dequantization [12] and 2-dimensional continuity dequantization. The first one (Uniform) has been discussed in Section 4. The second one (Var) used a learning-based method to dequantize data, aiming to minimize the distance between q and p, but still could not achieve continuity. The last one (Deq-2D) was implemented by us, constructing a continuous PDF on 2-dimensional data.

As shown in Fig.8 (a), on BJAQ, FACE outperformed Uniform and Var because it could generate more continuous dequantized data and make it easier for the NF model to fit. Besides, we could see that the accuracy of FACE is comparable to Deq-2D, which indicated that merely ensuring the continuity of marginal distribution was enough for the NF model to fit.

7.6.2 Sampled Data Sharing. We evaluated the sampled data sharing proposed in Section 6. FACE-noShare denoted the method without sampled data sharing, i.e., sampling iteratively for each query from scratch. We reported the average latency of FACE-noShare

Figure 8: Variance Evaluation.

Table 5: Evaluation of Data Updates.

20% Training		+20%	+20%	+20%	+20%
NoModelUpdate	Max	24.5	24.0	21	21.37
	95th	2.06	1.92	1.74	1.81
Inc-Training	Max	2.55	3.23	3.43	3.20
	95th	1.18	1.17	1.19	1.16
Retraining	Max	2.50	3.00	2.85	3.00
	95th	1.16	1.15	1.18	1.16

and FACE on 3 real-world datasets. As shown in Fig.8 (b), we improved the efficiency two times because FACE shared the sampled data with similar queries, and thus the convergence was fast.

7.7 Data Updates Evaluation

In this section, we studied the impact of data updates on FACE. Following [45], we partitioned Power into 5 parts on a time attribute, and then each partition came in order, i.e., each time we added 20% data into the training set. Given a query workload, we first trained on the first 20% data. The row of NoModelUpdate denoted that we trained on current arrived data, and directly estimate the cardinality of the workload when 20% data was added, without any model update. The row of Inc-Training denoted that when the 20% new data was added, we incrementally trained the model and estimated the query workload. Retraining denoted that we retrained the model from scratch when each partition came.

As shown in Table 5, for NoModelUpdate, the 95% and Max-quantiles were stable, indicating that FACE had a good generalization ability. Besides, by comparing Inc-Training with Retraining, we could see that FACE can adapt to data updates effectively.

8 CONCLUSION

In this paper, we propose FACE, a Flow-based novel cardinality estimator, which supports accurate estimation on different types of data. We design a spline dequantization method and utilize normalizing flow based model to learn the joint distribution of data. We also build an index to handle Like predicates for string attributes. For inference, we apply Monte Carlo integration to compute the cardinality and propose an acceleration algorithm. The results show that our method gains 50× performance improvement on accuracy.

REFERENCES

[1] Gleb Beliakov. 2005. Monotonicity Preserving Approximation of Multivariate Scattered Data. *BIT Numerical Mathematics* 45 (01 2005), 653–677. https://doi.org/10.1007/s10543-005-0028-x

[2] Laurent Dinh, David Krueger, and Yoshua Bengio. 2015. NICE: Non-linear Independent Components Estimation. In *ICLR*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1410.8516

[3] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek Narasayya, and Surajit Chaudhuri. 2019. Selectivity estimation for range predicates using lightweight models. *Proceedings of the VLDB Endowment* 12, 9 (2019), 1044–1057.

[4] Frederick N Fritsch and Ralph E Carlson. 1980. Monotone piecewise cubic interpolation. *SIAM J. Numer. Anal.* 17, 2 (1980), 238–246.

[5] Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. 2015. MADE: Masked Autoencoder for Distribution Estimation. In *ICML (JMLR Workshop and Conference Proceedings)*, Francis R. Bach and David M. Blei (Eds.), Vol. 37. JMLR.org, 881–889. http://proceedings.mlr.press/v37/germain15.html

[6] Zhabiz Gharibshah, Xingquan Zhu, Arthur Hainline, and Michael Conway. 2020. Deep Learning for User Interest and Response Prediction in Online Display Advertising. *Data Science and Engineering* 5, 1 (2020), 12–26. https://doi.org/10.1007/s41019-019-00115-y

[7] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *NIPS* 27 (2014), 2672–2680.

[8] LU Han and Larry L Schumaker. 1997. Fitting monotone surfaces to scattered data using C1 piecewise cubics. *SIAM journal on numerical analysis* 34, 2 (1997), 569–585.

[9] Shohedul Hasan, Saravanan Thirumuruganathan, Jees Augustine, Nick Koudas, and Gautam Das. 2020. Deep Learning Models for Selectivity Estimation of Multi-Attribute Queries. In *SIGMOD*. ACM, 1035–1050. https://doi.org/10.1145/3318464.3389741

[10] Max Heimel, Martin Kiefer, and Volker Markl. 2015. Self-Tuning, GPU-Accelerated Kernel Density Models for Multidimensional Selectivity Estimation. In *SIGMOD*, Timos K. Sellis, Susan B. Davidson, and Zachary G. Ives (Eds.). ACM, 1477–1492. https://doi.org/10.1145/2723372.2749438

[11] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *VLDB* 13, 7 (2020), 992–1005. http://www.vldb.org/pvldb/vol13/p992-hilprecht.pdf

[12] Jonathan Ho, Xi Chen, Aravind Srinivas, Yan Duan, and Pieter Abbeel. 2019. Flow++: Improving Flow-Based Generative Models with Variational Dequantization and Architecture Design. In *ICML*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 2722–2730. http://proceedings.mlr.press/v97/ho19a.html

[13] Emiel Hoogeboom, Taco S Cohen, and Jakub M Tomczak. 2020. Learning discrete distributions by dequantization. *arXiv preprint arXiv:2001.11235* (2020).

[14] Individual household electric power consumption data set. 2021. https://github.com/gpapamak/maf. Last accessed: 2021-09-14.

[15] Martin Kiefer, Max Heimel, Sebastian Breß, and Volker Markl. 2017. Estimating Join Selectivities using Bandwidth-Optimized Kernel Density Models. *VLDB* 10, 13 (2017), 2085–2096. https://doi.org/10.14778/3151106.3151112

[16] Diederik P. Kingma and Max Welling. 2014. Auto-Encoding Variational Bayes. In *ICLR*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1312.6114

[17] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*. www.cidrdb.org.

[18] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR*.

[19] Ivan Kobyzev, Simon Prince, and Marcus Brubaker. 2020. Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2020).

[20] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *VLDB* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[21] Viktor Leis, Bernhard Radke, Andrey Gubichev, Alfons Kemper, and Thomas Neumann. 2017. Cardinality Estimation Done Right: Index-Based Join Sampling. In *CIDR*. www.cidrdb.org.

[22] G Peter Lepage. 2021. Adaptive multidimensional integration: vegas enhanced. *J. Comput. Phys.* 439 (2021), 110386.

[23] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI Meets Database: AI4DB and DB4AI. In *SIGMOD*. 2859–2866. https://doi.org/10.1145/3448016.3457542

[24] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. Machine Learning for Databases. *Proc. VLDB Endow.* 14, 12 (2021), 3190–3193. http://www.vldb.org/pvldb/vol14/p3190-li.pdf

[25] Guoliang Li, Xuanhe Zhou, and Chengliang Chai. 2021. AI Meets Database: A Survey. In *TKDE*.

[26] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. openGauss: An Autonomous Database System. *Proc. VLDB Endow.* 14, 12 (2021), 3028–3041. http://www.vldb.org/pvldb/vol14/p3028-li.pdf

[27] Mingda Li, Hongzhi Wang, and Jianzhong Li. 2020. Mining Conditional Functional Dependency Rules on Big Data. *Big Data Mining and Analytics* 03, 01, Article 68 (2020), 16 pages.

[28] Thomas Müller, Brian McWilliams, Fabrice Rousselle, Markus Gross, and Jan Novák. 2019. Neural Importance Sampling. *ACM Trans. Graph.* 38, 5 (2019), 145:1–145:19. https://doi.org/10.1145/3341156

[29] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425* (2019).

[30] George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. 2021. Normalizing flows for probabilistic modeling and inference. *Journal of Machine Learning Research* 22, 57 (2021), 1–64.

[31] G Peter Lepage. 1978. A new algorithm for adaptive multidimensional integration. *J. Comput. Phys.* 27, 2 (1978), 192–203.

[32] Viswanath Poosala, Yannis E. Ioannidis, Peter J. Haas, and Eugene J. Shekita. 1996. Improved Histograms for Selectivity Estimation of Range Predicates. In *SIGMOD*, H. V. Jagadish and Inderpal Singh Mumick (Eds.). ACM Press, 294–305.

[33] PostgreSQL. 2021. https://www.postgresql.org/. Accessed: 2021-09-14.

[34] Danilo Jimenez Rezende and Shakir Mohamed. 2015. Variational Inference with Normalizing Flows. In *ICML (JMLR Workshop and Conference Proceedings)*, Vol. 37. JMLR.org, 1530–1538.

[35] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. 1979. Access Path Selection in a Relational Database Management System. In *SIGMOD*, Philip A. Bernstein (Ed.). ACM, 23–34. https://doi.org/10.1145/582095.582099

[36] Beijing Multi-Site Air-Quality Data Data Set. 2021. https://archive.ics.uci.edu/ml/datasets/Beijing+Multi-Site+Air-Quality+Data. Last accessed: 2021-09-14.

[37] Ji Sun and Guoliang Li. 2019. An End-to-End Learning-based Cost Estimator. *VLDB* 13, 3 (2019), 307–319. http://www.vldb.org/pvldb/vol13/p307-sun.pdf

[38] Ji Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation for Similarity Queries. In *SIGMOD*. 1745–1757. https://doi.org/10.1145/3448016.3452790

[39] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned Cardinality Estimation: A Design Space Exploration and A Comparative Evaluation. *VLDB* (2021).

[40] Lucas Theis, Aäron van den Oord, and Matthias Bethge. 2016. A note on the evaluation of generative models. In *ICLR*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1511.01844

[41] Shan Tian, Songsong Mo, Liwei Wang, and Zhiyong Peng. 2020. Deep Reinforcement Learning-Based Approach to Tackle Topic-Aware Influence Maximization. *Data Science and Engineering* 5, 1 (2020), 1–11. https://doi.org/10.1007/s41019-020-00117-1

[42] Benigno Uria, Iain Murray, and Hugo Larochelle. 2013. RNADE: The real-valued neural autoregressive density-estimator. In *NIPS*, Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (Eds.). 2175–2183. https://proceedings.neurips.cc/paper/2013/hash/53adaf494dc89ef7196d73636eb2451b-Abstract.html

[43] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready For Learned Cardinality Estimation? *VLDB* 14, 9 (2021), 1640–1654. http://www.vldb.org/pvldb/vol14/p1640-wang.pdf

[44] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Xi Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73. https://doi.org/10.14778/3421424.3421432

[45] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Peter Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *VLDB* 13, 3 (2019), 279–292. https://doi.org/10.14778/3368289.3368294

[46] Xiang Yu, Guoliang Li, Chengliang Chai, and Nan Tang. 2020. Reinforcement Learning with Tree-LSTM for Join Order Selection. In *ICDE*. IEEE, 1297–1308. https://doi.org/10.1109/ICDE48307.2020.00116

[47] Zhuoyue Zhao, Robert Christensen, Feifei Li, Xiao Hu, and Ke Yi. 2018. Random Sampling over Joins Revisited. In *SIGMOD*, Gautam Das, Christopher M. Jermaine, and Philip A. Bernstein (Eds.). ACM, 1525–1539. https://doi.org/10.1145/3183713.3183739

[48] Xuanhe Zhou, Ji Sun, Guoliang Li, and Jianhua Feng. 2020. Query Performance Prediction for Concurrent Queries using Graph Embedding. *Proc. VLDB Endow.* 13, 9 (2020), 1416–1428. https://doi.org/10.14778/3397230.3397238

[49] Rong Zhu, Ziniu Wu, Yuxing Han, Kai Zeng, Andreas Pfadler, Zhengping Qian, Jingren Zhou, and Bin Cui. 2021. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *VLDB* 14, 9 (2021), 1489–1502. http://www.vldb.org/pvldb/vol14/p1489-zhu.pdf

[50] Zachary M. Ziegler and Alexander M. Rush. 2019. Latent Normalizing Flows for Discrete Sequences. In *ICML (Proceedings of Machine Learning Research)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, 7673–7682.