The VLDB Journal (2024) 33:323–348
https://doi.org/10.1007/s00778-023-00808-x

REGULAR PAPER

Cardinality estimation using normalizing flow

Jiayi Wang[1] · **Chengliang Chai**[2] · **Jiabin Liu**[1] · **Guoliang Li**[1]

Received: 20 December 2022 / Revised: 4 June 2023 / Accepted: 28 July 2023 / Published online: 29 August 2023

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 for both sequential queries and parallel queries. 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 estimate the cardinality based on the `FACE` model. Fourth, we propose a grouping technique to process parallel queries. Fifth, we discuss how to support join queries. Experimental results show that our method significantly outperforms existing approaches in terms of *estimation accuracy* while keeping similar *latency* and *model size*.

Keywords Cardinality estimation · Query optimization · AI for DB

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 has widespread applications in the database community, such as query optimization, approximate query processing. In particular, 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 [43], sampling [29, 58] or kernel density-based methods [16, 22], cannot capture the column correlations, recently machine learning (ML)-based CE methods [12, 17, 25, 31–35, 46–48, 50, 55–57, 59] 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.

✉ Chengliang Chai
ccl@bit.edu.cn

✉ Guoliang Li
liguoliang@tsinghua.edu.cn

Jiayi Wang
jiayi-wa20@mails.tsinghua.edu.cn

Jiabin Liu
liujb19@tsinghua.edu.cn

[1] Department of Computer Science and Technology, Tsinghua University, Beijing, China

[2] Department of Computer Science and Technology, Beijing Institute of Technology, Beijing, China

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 [55, 60], 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 [25, 46] 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 [17, 55, 56] 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.

However, existing data-driven methods suffer from the following limitations. (1) Sum-product-network-based method [17] assumes different levels of independence between columns, based on which they recursively split rows and columns to learn the distribution, but the accuracy is low due to the assumption (cannot achieve ***O1***). Thus, the first challenge is how to capture the dependencies between different columns (**C1**). (2) Although `Naru` [55, 56], `DQM-D` [15] and `UAE` [54] 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 [15, 56], it leads to high training cost and high storage overhead (cannot achieve ***O3***). Even if `NeuroCard` [55] 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 [15, 55, 56] 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**).

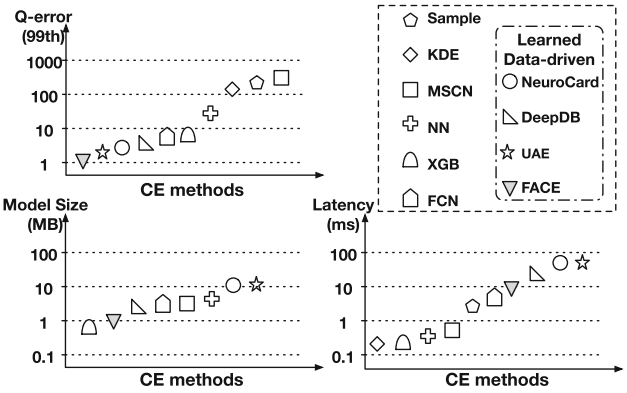

Fig. 1 Performance comparison of CE methods

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 [27, 38] 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 are 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***). In addition, we propose strategies to enable `FACE` to support join queries. Fourth, given the joint distribution learned by NF, we infer the cardinality by Monte Carlo integration over the distribution, which is computed through sampling data points from ranges in query predicates. In this situation, for an incoming query, it

can reuse the samples from previous similar queries, so as to improve the efficiency (addressing **C3**, for ***O2***). Finally, for parallel queries, we propose to leverage the query similarities to judiciously group similar ones such that queries in the same group can share sampled data points. In this way, the number of sampled data points can be greatly reduced, and thus the inference latency can be accelerated (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 adopt the Monte Carlo integration to conduct CE inference, where query similarities are considered to accelerate the process.
(4) We further accelerate CE for parallel queries by grouping them according to their similarities to avoid duplicated computation.
(5) We conducted extensive experiment on 4 datasets and compared with 12 baselines to show our superior performance.

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 [56]:

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

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 [15, 56], 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 Sect. 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, Eq. 1 can be written as:

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

2.2 Normalizing flow-based model

The joint data distribution is modeled via generative models, where GAN [13], VAE [24], Autoregressive [11] and Normalizing Flow (NF) [6, 42] 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 do not explicitly output the probability density, so it is intractable for them to estimate the cardinality. Although the autoregressive model [11] has been applied in CE recently, it still suffers from the large domain size problem, as discussed in Sect. 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 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 $\mathbf{u} \sim \pi(\mathbf{u})$ [38].

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

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

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

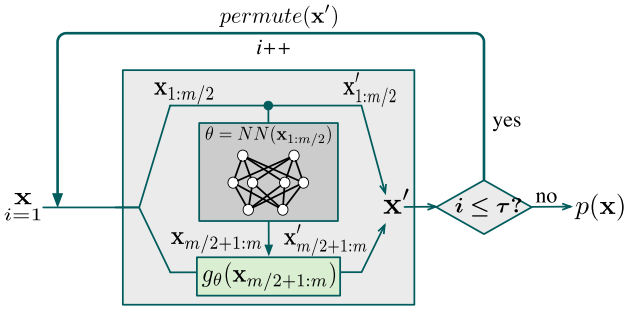

Fig. 2 An example of coupling-based flow models

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 [6, 36, 61] 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 hyper-parameter, say 5. Each coupling layer has the same input/output dimension, which is designed by the following steps:

- Divide input $\mathbf{x}$ into two equal parts: $[\mathbf{x}_{1:d}, \mathbf{x}_{d+1:m}]$, $d = \frac{m}{2}$.
- 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. In the training step, they collect a pool of queries with their real cardinalities as labels and then train a model to map a query to its cardinality. For inference, query is encoded and then fed into the regression model. Different models are used, including fully connected neural networks [7, 37], CNN [26],RNN [37, 46]. In general, query-driven CE methods need a large amount of training data, i.e., queries. If the query distribution shifts, the model is likely to behave poorly. Therefore, query-driven approaches are expensive and not generalizable enough.
Data-driven learned CE methods. They learn the joint data distribution with different models. When inference, they use the model to infer the probability of tuples satisfying the query predicates.

(1) Normalizing flow model [52]. The conference version of this paper leverages the Normalizing Flow based model to learn a continuous joint distribution for relational data. To estimate the cardinality of a query, it samples some data points from the ranges in query predicates and applies Monte Carlo integration over the learned joint distribution. In this paper, we extend it to better support parallel queries and multi-table queries.
(2) Sum-Product network [17]. The 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. It is inaccurate because the independence assumption is made.
(3) Autoregressive models [15, 55, 56]. 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 [56] and DQM-D [15] 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 [55] 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

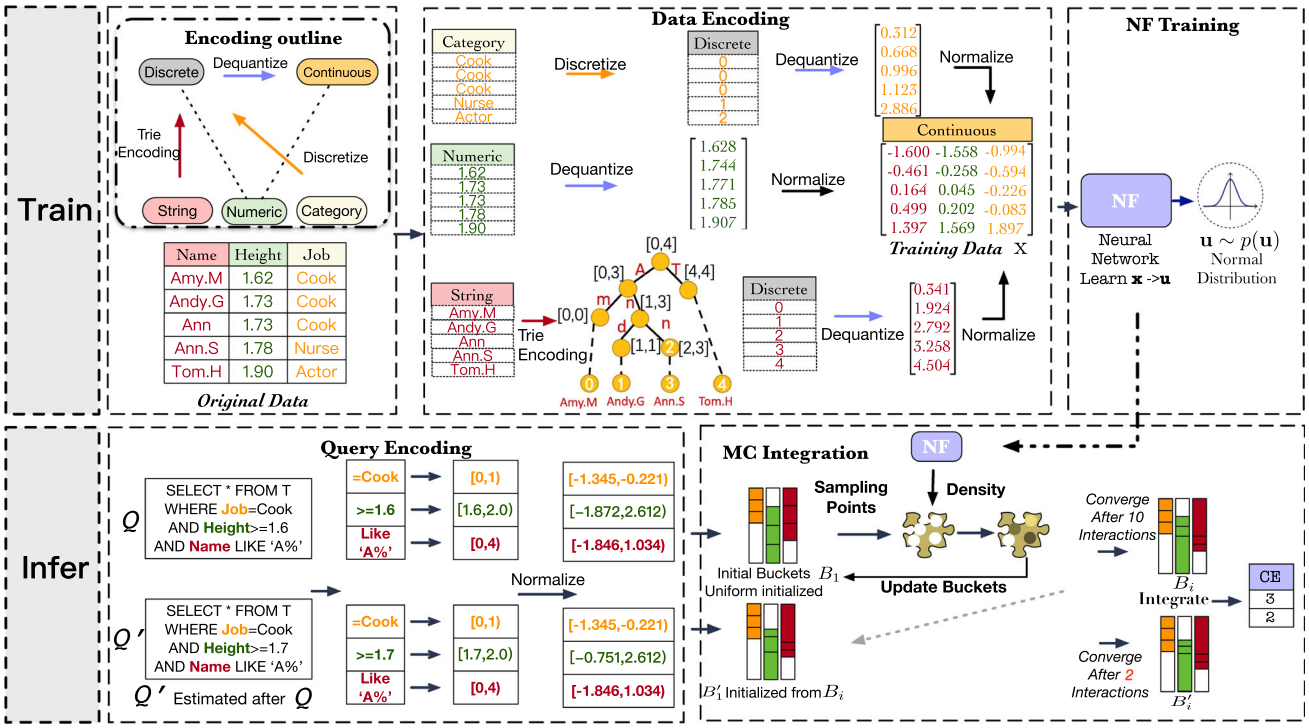

Fig. 3 The framework of FACE

idea of using NF (Sect. 3.1), and then the overall architecture (Fig. 3) of FACE (including training (Sect. 3.2) and inference (Sect. 3.3)). Finally, we summarize how FACE can estimate join queries in Sect. 3.4.

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 encodes appropriately and generates the encoded data that can be fed into the NF model (Sect. 3.2). After training, we can compute the probability density using the NF model, i.e., $p(\mathbf{x})$.

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

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

Note that not all predicates are range predicates. Therefore, to apply Eq. 4, we transfer other predicates to ranges (see Sect. 3.3). Then, as the inference part in Fig. 3 shows, we sample some data points from these ranges (see Sect. 6), call NF model to estimate the probability density of them and finally compute the estimated cardinality using Monte Carlo (MC) integration [30].

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 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 Sect. 4) and build an index to tackle `Like` predicates with string data (see Sect. 5). (3) The repetitive sampling is time-consuming in the inference step, so an acceleration method is proposed in Sect. 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 [15, 56] 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 ($E(a_i) \rightarrow w$), e.g., $E(Cook) \rightarrow 0$. However, if we fit discrete data directly with a continuous density model, it will produce a degenerate solution that places all probability mass on the discrete data points. Therefore, we use the *dequantization* [19, 49] method that adds noise to discrete data over the width of each discrete bin. This method makes data more 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(`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., $\int_0^1 p(x)\mathrm{d}x = 0.6$, where $p(x)$ is learned by NF. The dequantization technique is significant in accuracy improvement for NF models, so in Sect. 4, we propose an 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. For continuous data, intuitively, we feed it into NF with no processing. However, any data in a computer are represented by a finite number of bits, 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 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 ones. 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 add `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 using maximum likelihood estimation.

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 categorical, we encode it to the same discrete value as the encoding in the training phase i.e., $E(a_i) \rightarrow w$. Then the range is constructed by $[w, w + bin)$, where bin is the 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 asso-

ciated 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 Sect. 5).

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 Eq. 4. Unfortunately, the integration is infeasible to compute, because it has no closed-form solution. Thus, MC integration [39] 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 [30, 39] 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, the 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.

Accelerate parallel queries. In real scenarios, hundreds of queries may come simultaneously in peak hours. Suppose every single query has to sample K data points for MC integration, n queries lead to nK data points. When n is large, nK data points lead to numerous computations, thus bringing high inference latency. Note that similar queries have similar query predicates and thus they can use similar sampled data points. To this end, we can share these data points among similar parallel queries.

Accordingly, we can reduce the total number of sampled data points and improve the overall efficiency.

In Sect. 6, we introduce how to compute the query similarity. We then illustrate how to accelerate the inference of sequential queries using buckets. After that, in Sect. 7, we propose how to group parallel queries into similar groups, within which queries share sampled data points to improve efficiency.

3.4 Supporting joins

`FACE` can be extended to support join queries with two strategies: *Single model* and *Multi-models*.

Single model applies the technique in `NeuroCard` [55] that leverages one estimator to learn the distribution of the full-outer-join table to support joins. However, full outer join can differ in distribution with the join results of queries because of duplicates and NULL values. To address this, following [17, 55], we need to add additional columns to record these duplicates and NULL values, so as to correct the distribution considering the queries. For inference, we will leverage the values in additional columns to correct the probability densities.

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 propose *Multi-models* to estimate cardinalities with multiple models. Specifically, 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. We will describe *Multi-Models* in detail in Sect. 8.

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 (Sect. 3.2.1) and use NF to fit them, we will derive a probability density function (PDF) as shown in Fig. 4a. This way has two limitations. On the one hand, fitting a continuous model to discrete data will produce a degraded solution [18] 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 p because the integral interval is unknown. Therefore, dequantization has to be applied.

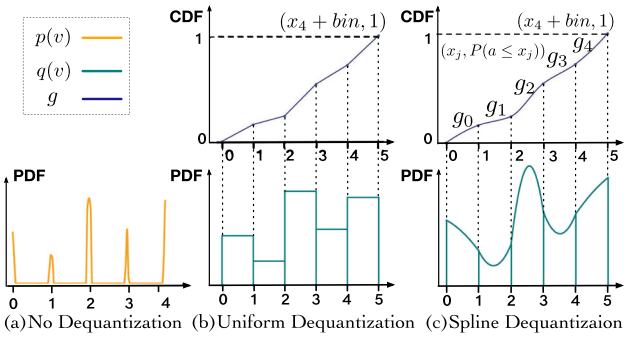

Fig. 4 Visualization of dequantization methods

Dequantization distribution. As discussed in Sect. 3.2.1, dequantization is utilized to add noise on discrete data so that NF can learn the continuous probability distribution better. 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 bin $[x, x + bin)$, so the integration over the bin precisely captures the probability of x.

Then the noise is generated based on q, and each discrete value becomes dequantized $v = x + u$ (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 Sect. 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 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 [49] 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 $bin = 1$. Figure 4b 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 [18, 19] 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 derivative of the CDF, q is naturally continuous.

For example, as shown in Fig. 4b, the CDF of the uniform dequantization is not continuously differentiable, so q is not continuous and the generated dequantized data are 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. 4c. Second, we use Monotone Piecewise Cubic Spline Interpolation [9] 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 threefold. (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. 4c, 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, 14]. As it is prohibitively

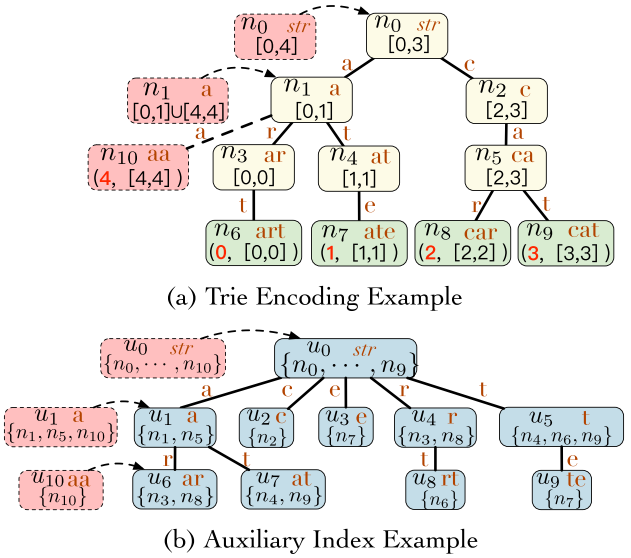

Fig. 5 String encoding example

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. For example, given $A_i = \{\texttt{art}, \texttt{ate}, \texttt{car}, \texttt{cat}\}$, we can build a trie $\mathcal{T}$ as shown in Fig. 5a. 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 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 the same way as numerical data. Next, we discuss how to conduct inference.

Inference of prefix-based predicates. For the prefix-based predicates, i.e., `str%`, if there exists a node with $n.str = \texttt{str}$, 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 the 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, $A_i' = \{\texttt{tra}, \texttt{eta}, \texttt{rac}, \texttt{tac}\}$. 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, $u_7.str = \texttt{at}$, and thus $u_7.s = \{n_4, n_9\}$ because $n_4.str = \texttt{at}$ and $n_9.str = \texttt{cat}$. To build $\mathcal{T}_a$, we start with the root that $u_0.str = \texttt{NULL}$. 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 $\mathcal{T}$. We repeat the above steps iteratively. To accelerate, we propose a pruning strategy. We limit the height of the tree to H. 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**). Figure 5 (b) shows the example with $H = 3$.

[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.

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. 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 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 Sect. 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. 5b). $\mathcal{T}_a$ 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) [30, 39] to conduct the inference. We first introduce its motivation and the basic solution in Sect. 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 (Sect. 6.2).

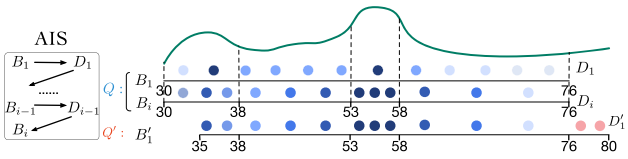

Fig. 6 An example of adaptive importance sampling

6.1 Adaptive importance sampling

Basic idea. For inference, as discussed in Sect. 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 (Eq. 4). However, as shown in Fig. 6, the probability density function p is always too complicated to integrate, so MC integration [30, 39] is always applied to approximate the result.

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

AIS. AIS [30, 39] is proposed to split each range R_i[2] into a sequence of successive buckets $B = [b_1, b_2, \ldots, b_{|B|}]$, 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. 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 $D_1, D_1, \ldots D_i$ to compute a weighted MC integration, 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,

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

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 Sect. 2.1). Given another query Q', we define $\text{sim}(Q, Q') = \frac{1}{m}\sum_{i=1}^{m}\frac{|R_i \cap R'_i|}{|R_i \cup R'_i|}$. The similarity score is in [0, 1]. The higher $\text{sim}(Q, Q')$ is, the more similar Q and Q' are.

Share with subsequent queries. In many cases, queries come in the form of streaming data. If the CE of each query requires to 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. Specifically, given a new coming query (e.g., Q' in Fig. 6), we find the most similar query among all estimated queries, 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. 6c, 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. On the one hand, if there exist data points of D_i that do not lie in 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 are required in R'_i.

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 Inference acceleration for parallel queries

In some real scenarios, hundreds of queries may come simultaneously in peak hours. Suppose every single query has to sample K data points for MC integration, n queries lead to nK data points. When n is large, nK data points lead to numerous computations, thus bringing high latency for estimating their cardinalities. Fortunately, given the observation that similar queries are always associated with similar sampled data points, we can share these data points among similar parallel queries such that the total number of data points to be sampled is reduced, and thus the overall efficiency can be improved. Based on this idea, we formulate the problem of inference acceleration for parallel queries as how to group similar queries to achieve the largest efficiency improvement. We first show the overall solution in Sect. 7.1. Then, we formally define the optimization problem on how to group queries, and then prove its NP-hardness and inapproximability (Sect. 7.2), and finally introduce our solution (Sect. 7.3).

7.1 Overview

Consider n queries $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$ that come simultaneously[3] over a relation T with m attribute.[4] Each Q_i is associated with an attribute subset $\mathcal{A}_i$ of the m attributes. In addition, each query Q can be represented as m ranges $R_1, R_2, \ldots, R_m$ (Sect. 2). For ease of representation, we use $Q.R_k, k \in [1, m]$ to denote the m ranges of query Q and $Q.B_k, k \in [1, m]$ to denote the corresponding buckets.

As discussed above, we can partition $\mathcal{Q}$ into g groups $\mathcal{G} = \{G_1, G_2, \ldots, G_g\}$ such that each group contains similar queries that can well share sampled data points. For each group G_i, we should have sampled $K \cdot |G_i|$ data points for estimation, but under this group-based optimization, we can sample a small batch (suppose that the size is denoted by b, $b \ll K \cdot |G_i|$) of data points to be shared by these $|G_i|$ queries. In this way, the total number of sampled data points is reduced, thus improving the efficiency of CE for $\mathcal{Q}$.

Generate samples for each group. Suppose that we have already partitioned the queries into groups. Then, for each group G_i, since none of the queries in G_i has already been estimated in advance, we have to first generate some sampled data points for data sharing, so as to make the queries in group G_i share these data points better. To be specific, we first simulate a proxy query S based on all the $Q_j \in G_i$ such that S is similar to each $Q_j \in G_i$. We set S as $S.R_k = \cup_{Q_j \in G_i} Q_j.R_k, k \in [1, m]$. Since queries within G_i are similar to each other, S is also similar to each $Q_j \in G_i$. Therefore, the data points of S can be well shared within G_i. Besides, since the query range of each $Q_j \in G_i$ is completely covered by $S.R$, each $Q_j \in G_i$ can leverage the data points of S without generating other data points as described in Sect. 6.2. Therefore, we just need to sample b data points

[3] We approximately assume that the queries appearing within a small time window (e.g., 1ms) are coming simultaneously.

[4] Our method can support multiple relations with joins. We use one relation here for ease of representation.

Algorithm 1: Parallel CE Acceleration (PCEA)

Input: Query set $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$.
Output: Estimated cardinalities for $\mathcal{Q}$.

```
1  /* Partition Q into groups */
2  G = CAGroup (Q)
3  for i = 1 to |G| do
4      Create S, where S.R_k = ∪_{Q_j ∈ G_i} Q_j.R_k, k ∈ [1, m];
5      while AIS is not converged do
6          Sample b data points for S;
7          for query Q_j ∈ G_i do
8              Select data points in Q_j.R from the b samples;
9              Update Q_j.B by selected data points;
10     Compute car(Q_j) for Q_j ∈ G_i;
11 return {car(Q_1), car(Q_2), ..., car(Q_n)};
```

to accurately estimate the cardinalities of queries in G_i and thus achieving high efficiency.

Overview. Next, we overview our parallel CE acceleration algorithm (PCEA) in Algorithm 1. PCEA takes the query set $\mathcal{Q}$ as input, partitions $\mathcal{Q}$ into groups (line 2) and shares samples within each group (lines 3–9) and finally computes the estimated cardinalities for queries in $\mathcal{Q}$ (lines 10, 11). Note that in this part, we only focus on how to accelerate CE for $\mathcal{Q}$ given $\mathcal{G}$, and leave how to partition $\mathcal{Q}$ into $\mathcal{G}$ to Sect. 7.3.

Partition $\mathcal{Q}$. PCEA first partitions $\mathcal{Q}$ into groups $\mathcal{G}$ according to the similarities between queries by CAGroup algorithm (line 2), which will be introduced in Sect. 7.3.

Data sharing. Next, PCEA performs AIS for each group of queries by sharing data points (line 3–9). For group G_i, S is first created to generate data points to be shared by $Q_j \in G_i$. As discussed above, the query range of S is the union of $Q_j.R_k, k \in [1, m]$ for all $Q_j \in G_i$. For example, in Fig. 7, the union of $Q_1.R_1$ ([30, 76]) and $Q_2.R_1$ ([35, 80]) over the column Age leads to a query S with $S.R_1 = [30, 80]$. Then for each iteration of AIS, S will generate b data points and use the NF model to calculate their densities. After that, from the b data points, each $Q_j \in G_i$ will select the data points falling into its own ranges $Q_j.R_k$, and next update the corresponding buckets $Q_j.B_k$ (see Sect. 6). For example, in Fig. 7, S generated 7 data points, 33, 45, 54, 57, 58, 65, 79. Then Q_1 used 6 data points except 79 to update its buckets $Q1.B$, because only 79 is outside $Q1.R$ ([30, 76]). We can see that although the total number of data points is less (from $|G_i| \cdot K$ to b), the number of data points that each query can leverage just decreases slightly, which is close to b. Thus PCEA can achieve efficient and accurate estimation.

CE for each group. The above data sharing process is iterated for several times until AIS of each $Q_j \in G_i$ converges. After that, for each $Q_j \in G_i$, Q_j uses the data points selected in each iteration and $Q_j.B$ to compute a weighted MC integration so as to obtain $car(Q_j)$ (see Sect. 6.1). After all groups of queries G_i have been estimated, we can get the estimated cardinalities of the whole $\mathcal{Q}$ (line 11). Thus,

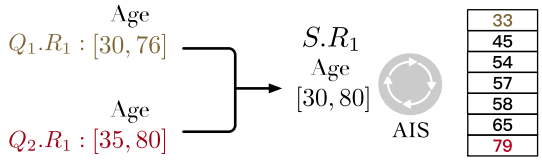

Fig. 7 An example of parallel sampled data sharing

the only problem that remains unsolved in PCEA is how to group queries (line 2), which will be discussed next.

7.2 Query grouping problem

In general, query grouping is to partition the n parallel queries into g disjoint groups G_i, $G_i \subseteq \mathcal{Q}$, $1 \leq i \leq g$. Apparently, we expect that the grouping strategy leads to both high accuracy and high efficiency CE. To ensure accuracy, any two queries Q_i and Q_j in the same group should have large similarity in order to well share data points with each other. To ensure efficiency, since the total number of sampled data points is proportional to the number of groups, g should be as small as possible as long as the data points within each group are similar.

Therefore, we formally define the **optimal queries grouping** (OQG) problem as follows:

$$\mathcal{G}^* = \arg\min |\mathcal{G}|, \text{ s.t.} \\ \forall G_i \in \mathcal{G}, \forall Q, Q' \in G_i, \text{sim}(Q, Q') \geq \epsilon \tag{5}$$

where ϵ is the threshold, and the constraint $\text{sim}(Q, Q') \geq \epsilon$ indicates that the similarity of any pair of queries in each group is no smaller than ϵ.

Now, we prove the NP-hardness of OQG by reducing the classic NP-hard Vertex Clique Cover (VCC) problem [10] to it. For an undirected, unweighted, simple graph $G = (V, E)$, VCC is to partition V into as few cliques as possible, where a clique C is a subset of V within which every two vertices are adjacent.

Theorem 1 *The OQG problem is NP-hard.*

Proof Consider a graph $G = (V, E)$, where V contains n vertices $\{v_1, v_2, \ldots, v_n\}$ and vertex v_i denotes query Q_i in $\mathcal{Q}$. There exist an edge connecting two vertices $v_i, v_j, i \neq j$ if and only if $\text{sim}(Q_i, Q_j) \geq \epsilon$. We next prove that OQG on $\mathcal{Q}$ can be reduced from VCC on G.

We first prove that there is a one-to-one mapping between cliques in G and groups in $\mathcal{Q}$. Since there is a one-to-one mapping between vertex v_i and query Q_i, a set of vertices $V' \subseteq V$ in G can be one-to-one mapped to a set of queries $\mathcal{Q}' \subseteq \mathcal{Q}$ and vice versa.

If V' is a clique, every two vertices in V' should be adjacent. Therefore, the corresponding two queries Q and Q' of the two vertices should satisfy $\text{sim}(Q, Q') \geq \epsilon$. This means

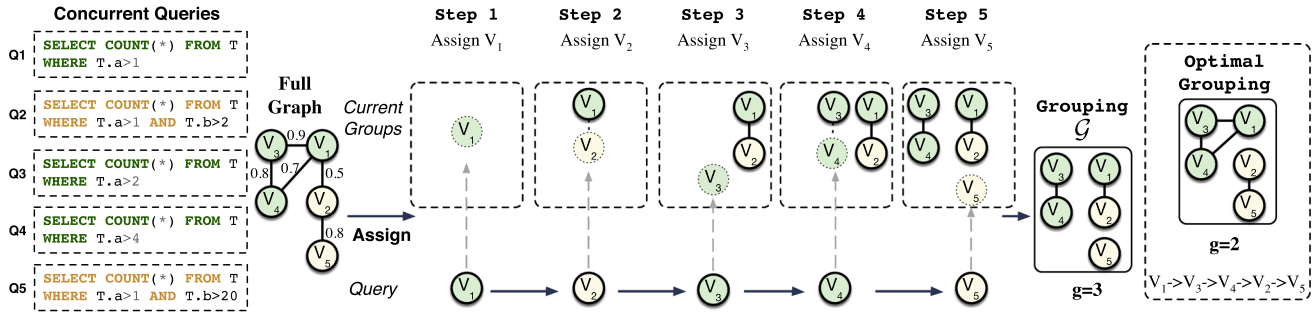

Fig. 8 An example of Greedy

that for any two queries Q and Q' in $\mathcal{Q}'$, $\text{sim}(Q, Q') \geq \epsilon$. Therefore, $\mathcal{Q}'$ should be a group.

On the other hand, if $\mathcal{Q}'$ is a group, every two queries Q and Q' in $\mathcal{Q}'$ should satisfy $\text{sim}(Q, Q') \geq \epsilon$. Therefore, the corresponding two vertices of Q and Q' in G should be adjacent. This means that any two vertices in V' are adjacent. Therefore, V' should be a clique. Hence, we proved that there is a one-to-one mapping between cliques in G and groups in $\mathcal{Q}$.

With this mapping, the VCC problem that partitions V into as few cliques as possible can be reduced to partition the queries in $\mathcal{Q}$ into as few groups as possible, which is exactly the OQG problem. Thus, we successfully reduce VCC to OQG. Since VCC is an NP-hard problem [21], OQG is also an NP-hard problem. □

Theorem 2 *For all $\alpha > 0$, it is NP-hard to approximate the OQG problem to within $n^{1-\alpha}$.*

Proof Since we have proved above that OQG is equivalent to VCC, which is an NP-hard problem that has been proved to satisfy Theorem 2 [62]. Therefore, OQG also satisfies Theorem 2. □

Since we have proved the equivalence between queries (groups) in $\mathcal{Q}$ and vertices (cliques) in G, for ease of illustration, we will next use these notations interchangeably. Next, we will propose a heuristic method to solve the OQG problem efficiently and effectively.

7.3 Group method

Naive solution. A basic Greedy heuristic is to straightforwardly iterate queries in $\mathcal{Q}$. In each iteration, we add a query into an existing group or create a new group for the query [4]. To be specific, suppose that the current iterated query is Q. Greedy checks whether Q can be added to any existing group, where Q can be added to a group G_i if $\forall Q' \in G_i, \text{sim}(Q, Q') \geq \epsilon$. If there exist such groups, we pick the first one for Q to be added into. Otherwise, Greedy creates a new group for Q.

Although Greedy is practical and easy to implement, it cannot achieve a small number of groups. To illustrate this, we give a simple example with 5 queries $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_5\}$ in Fig. 8 with $\epsilon = 0.5$. Suppose that Greedy iterates $\mathcal{Q}$ in the order of Q_1, Q_2, Q_3, Q_4, Q_5. First, Greedy adds Q_1 and creates the first new group $G_1 = \{Q_1\}$. Then, since Q_2 exactly satisfies the similarity constraint with Q_1, *i.e.,* $\text{sim}(Q_1, Q_2) = \epsilon = 0.5$, Greedy can add Q_2 to G_1. Next, since $\text{sim}(Q_3, Q_2) < 0.5$ and $Q_2 \in G_1$, Greedy can not add Q_3 to G_1, so it has to create a new group G_2 for Q_3. Similarly, Greedy next adds Q_4 to G_2 and creates a new group G_3 for Q_5, finally leading to 3 groups. However, as shown in Fig. 8, the optimal grouping strategy only has 2 groups.

Greedy is not the optimal because when tackling Q_2, it is added into the same group with Q_1. Although they satisfy the similarity constraint, their similarity is exactly the threshold (*i.e.,* 0.5). This makes the group diverse and thus leaves very limited room for other queries to be added into. On the contrary, if we swap Q_2 and Q_3 in the iteration order, *i.e.,* Q_1, Q_3, Q_2, Q_4, Q_5, we can obtain the optimal number of groups in the end as shown in Fig. 8. Because in this order, Q_3, a query more similar to Q_1 compared to Q_2, is added to G_1, which enables Q_4 to be added into G_1. Hence, we come to the intuition that iterating queries in an order that any two adjacent queries are highly similar allows each group to accommodate more queries, thus leading to less number of groups.

To address this, Iterated Greedy (IG) [5] optimizes Greedy by repeatedly performing Greedy with different iteration orders of $\mathcal{Q}$ and finally outputs the $\mathcal{G}$ with the minimum number of groups found in the above process. IG can find a $\mathcal{G}$ with fewer groups compared to Greedy. However, IG has to perform Greedy many times to discover a better grouping strategy, which is time-consuming and the time complexity is $O(n^4)$. There are also a line of works that optimize VCC for graphs with special structures like planar graphs [2, 3, 45]. However, these methods are rather inefficient for general graphs.

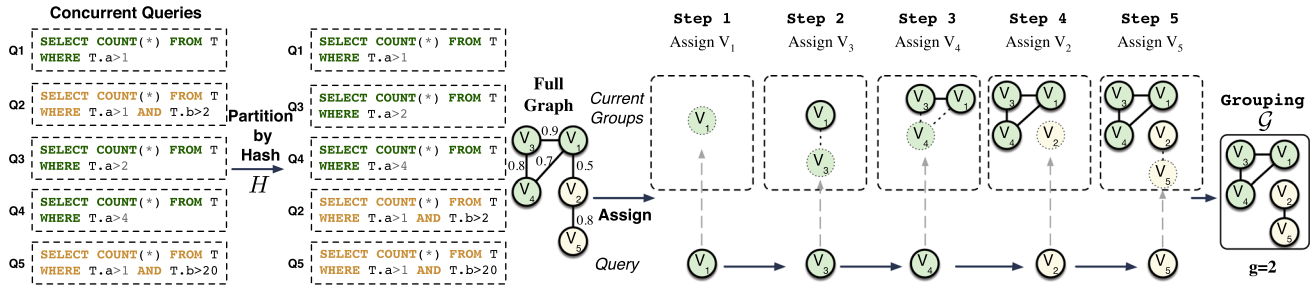

Fig. 9 An example of CAGroup

Algorithm 2: CAGroup

Input: Query set $\mathcal{Q} = \{Q_1, Q_2, \ldots, Q_n\}$, query similarity threshold ϵ.
Output: Grouping $\mathcal{G}$

1 $\mathcal{G} = \emptyset$;
2 Hash $\mathcal{Q}$ into slots by $h(\mathcal{A}_i)$;
3 **for** *each hash slot* $\mathcal{S}(\mathbf{h})$ **do**
4 **for** *query* $Q_i \in \mathcal{S}(\mathbf{h})$ **do**
5 Sort $\mathcal{G}$ by $\texttt{Gsim}(Q_i, G_j)$ in descending order;
6 $flag$ = False;
7 **for** $G_j \in \mathcal{G}$ **do**
8 **if** $\texttt{sim}(Q_i, Q_k) \geq \epsilon, \forall Q_k \in G_j$ **then**
9 Add Q_i to G_j;
10 $flag$ = True;
11 Break;
12 **if** $flag$ = *False* **then**
13 Add $\{Q_i\}$ to $\mathcal{G}$;
14 **return** $\mathcal{G}$;

Basic idea. As discussed above, intuitively, an ideal iteration order should satisfy that any two adjacent queries Q_i and Q_j in the order are highly similar, but it is too costly to attempt all the $n!$ orders. Fortunately, we can observe that queries with similar $\mathcal{A}$ are likely to have high similarity. Based on this, our basic idea is to first hash the queries in $\mathcal{Q}$ to different slots by a hash function $h(\mathcal{A})$. Thus the queries in each slot, *i.e.,* having the same hash value $h(\mathcal{A})$, are highly similar to each other. In this way, we iterate queries slot by slot and handle the queries in each slot one by one so that we can obtain an order where most adjacent queries are highly similar. In addition, for each query Q_i, if there are multiple groups that Q_i can be added to, we choose to add it into the group where all queries are similar to Q_i. This allows for lower query diversity within each group, thus leaving more room to accommodate other queries. Finally, when queries in all slots have been iterated, we obtain a near-optimal grouping $\mathcal{G}$ for $\mathcal{Q}$.

Hash function $h(\mathcal{A})$. We use a simple hash function $h(\mathcal{A})$ that takes $\mathcal{A}$ as the key and an m-dimensional $0-1$ vector as the hash value. To be specific, the j-th dimension of $h(\mathcal{A})$ is taken as 1 if $A_j \in \mathcal{A}$, otherwise 0. For ease of description, we use $\mathcal{S}(\mathbf{h})$ to denote the set of queries in the same slot with hash value $h(\mathcal{A}) = \mathbf{h}$. For example, in Fig. 9, there are 5 queries over a table with 2 attributes $\{a, b\}$. Since $\mathcal{A}_1 = \{a\}$, the hash value $h(\mathcal{A}_1) = 10$. Because $\mathcal{A}_1 = \mathcal{A}_3 = \mathcal{A}_4 = \{a\}$, Q_1, Q_3, Q_4 have the same hash value $\mathbf{h} = 10$ and are hashed to the same slot $\mathcal{S}(10) = \{Q_1, Q_3, Q_4\}$.

Our solution. Based on the above idea, we propose an efficient yet effective heuristic algorithm CAGroup (Column-Aware Grouping) to solve OQG. As shown in Algorithm 2, CAGroup takes $\mathcal{Q}$ and the query similarity threshold ϵ (default as 0.5) as input, and finally outputs the groups $\mathcal{G}$.

Specifically, in CAGroup, we first initialize $\mathcal{G}$ as $\emptyset$. (line 1). Then, we hash queries in $\mathcal{Q}$ to different slots $\mathcal{S}(h(\mathcal{A}_i))$ (line 2). Next, we iterate through the slots and handle queries Q_i in each slot $\mathcal{S}(\mathbf{h})$ one by one (lines 3–13). Since we expect each Q_i to be added to the group in which all queries are similar to Q_i, we can define a query-group similarity to measure Q_i and each group $G_j \in \mathcal{G}$.

To this end, we first encode each G_j into an m-dimensional vector $\mathbf{e}_j$ as the average of $h(\mathcal{A}_k)$ for all $Q_k \in G_j$, *i.e.,* $\mathbf{e}_j = \frac{\sum_{Q_k \in G_j} h(\mathcal{A}_k)}{|G_j|}$. The vector $\mathbf{e}_j$ denotes the distribution of $\mathcal{A}_k$ for all the queries $Q_k \in G_j$. For example, the t-th dimension of $\mathbf{e}_j$ indicates the proportion of query $Q_k \in G_j$ that satisfies $A_t \in \mathcal{A}_k$. Therefore, we can use $\texttt{Gsim}(Q_i, G_j) = 1 - \frac{\|h(\mathcal{A}_i) - \mathbf{e}_j\|_1}{m}$ to reflect the similarity between Q_i and queries in G_j. Gsim is between [0, 1]. The larger $\texttt{Gsim}(Q_i, G_j)$ is, the more similar Q_i and queries in G_j will be.

With the help of Gsim, we sort the groups in $\mathcal{G}$ by the value of $\texttt{Gsim}(Q_i, G_j)$ in descending order (line 5) and then iterate them to add Q_i to the first group G_j that satisfies $\texttt{sim}(Q_i, Q_k) \geq \epsilon$ for all $Q_k \in G_j$ (line 8). This allows us to add Q_i to the group most similar to it (line 9), thus leaving more room to accommodate other queries. If Q_i cannot be added to any group (line 12), we add a new group $\{Q_i\}$ to $\mathcal{G}$ (line 13). Finally, after we have iteratively handled all the $Q_i \in \mathcal{Q}$, we can get a near-optimal $\mathcal{G}$ for $\mathcal{Q}$ (line 14).

Example. Figure 9 shows the same example of 5 queries that have been used for Greedy with $\epsilon = 0.5$. First, CAGroup hashes $\mathcal{Q}$ into two slots according to $h(\mathcal{A})$, *i.e.,* $\mathcal{S}(h(\mathcal{A}_1))$ in green and $\mathcal{S}(h(\mathcal{A}_2))$ in yellow. After that, CAGroup first handles $Q \in \mathcal{S}(h(\mathcal{A}_1))$, and then handles $Q \in \mathcal{S}(h(\mathcal{A}_2))$,

that is, iterates $\mathcal{Q}$ in the order of Q_1, Q_3, Q_4, Q_2, Q_5. In the first 3 steps, Q_1, Q_3, Q_4 are added one by one to the same group G_1, because CAGroup makes G_1 less diverse (queries have high similarity with each other), thus leaving larger room for more queries. Then, in step 4, CAGroup tries to add Q_2 to group G_1. However, because $\texttt{sim}(Q_2, Q_1) < 0.5$ and $Q_1 \in G_1$, Q_2 cannot be added to G_1. Therefore, CAGroup makes Q_2 itself as a new group G_2. After that, in step 5, since $\texttt{Gsim}(Q_5, G_2) = 1 > \texttt{Gsim}(Q_5, G_1)$, CAGroup first checks whether Q_5 can be added to group $G_2 = \{Q_2\}$. Since $\text{sim}(Q_5, Q_2) \geq 0.5$, it adds Q_5 to G_2 and finally outputs $\mathcal{G}$. $\mathcal{G}$ has the optimal number of groups (*i.e.,* 2) because CAGroup successfully reorders the queries in $\mathcal{Q}$ and adds each query to the most similar group to make each group accommodate more queries.

Complexity analysis. Recap that there are n queries in total over a table with m columns. To compute $\mathcal{G}$, we handle the n queries $Q_i \in \mathcal{Q}$ one by one (lines 3–13). Each time, groups in $\mathcal{G}$ are first sorted by the value of $\texttt{Gsim}(Q_i, G_j)$, which can be done in $O(|\mathcal{G}|m)$ using bucket sort. After that, we check whether Q_i can be added to any group. In the worst case, we need to check whether $\texttt{sim}(Q_i, Q) \geq \epsilon$ for all the other $Q \in \mathcal{Q}$. Since each $\texttt{sim}(Q_i, Q)$ can be computed in $O(m)$, the total time is $O(nm)$. Finally, Q_i is added to an existing group or used to create a new group, both of which can be done in $O(m)$. Therefore, since $|\mathcal{G}| \leq n$, the total time complexity of CAGroup is $O(n^2m)$, much faster than IG while achieving comparable performance.

8 Supporting joins with multi-models

In Sect. 3.4, we discussed two methods to support joins, i.e., *Single model* which uses a single model to support all tables and *Multi-models* which uses multiple models where each model supports a join template of multiple tables.

The *single model* builds a single model, but the full-join table may be very sparse and the trained model may not be effective for different queries. To address this issue, 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 of the query. However it is rather expensive to enumerate all possible joins and build a model for each join. To alleviate this issue, we use the historical queries to generate query templates (a query template is a join query by removing all predicates and only keeping the join structure), among which frequent query templates (e.g., frequency > 10) are leveraged for training. If there is no historical queries, we can use the primary-key/foreign-key to generate the templates. If a query is not covered by an existing template, we use each single-table model to estimate the cardinality of each table and then combine them together using independence hypothesis.

Table 1 Real datasets

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

To summarize, the advantage of the multi-models method is that it can provide more fine-grained estimation than the *single model* for queries covered by existing templates. However, this method needs additional join template information, and it may consume larger memory when the number of models is large. In Sects. 9.5–9.7, we will evaluate the above two strategies on the benchmark JOB-light [25] with queries over multiple tables.

9 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 Sects. 9.1–9.8. Then we evaluated our proposed techniques in Sects. 9.9–9.10.

9.1 Experimental settings

Dataset. We used three widely used real-world datasets [7, 15, 28], and TPC-H, a widely used benchmark. Table 1 shows the datasets statistics. 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 [20] is a household electric power consumption data. It has large domain sizes in all columns (each $\approx$ 2 M) and all columns are numerical data. (2)IMDB [28] 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 [58] 4,740,297 tuples from the final result uniformly. The domain size of IMDB varies a lot, from 2 to 1 M. IMDB contains highly skewed attributes, *e.g.,* country_code. (3)BJAQ [44] includes hourly air pollutants data of Beijing, which has medium domain sizes (1K-2K). (4)TPC-H is a commonly used synthetic benchmark dataset, which contains 22 query templates. We used scale factor of 10 to generate 10 GB data and used the query templates to generate 2000 different queries.

Baselines. We compared FACE with a variety of typical CE algorithms, including:

Table 2 Q-errors, latency (ms) and model size (MB) on 4 datasets

Estimator	50th	95th	99th	Max	Latency	Model size
(a) Power						
PG	1.38	15.6	118	$3 \cdot 10^5$	1.25	**0.92**
Sample	1.04	1.97	150	722	2.07	–
MHIST	5.10	135	383	$2 \cdot 10^5$	2070	11
KDE	1.36	18.2	119	$1 \cdot 10^3$	**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
FCN	1.08	2.46	7.66	225	3.64	1.5
DeepDB	1.06	1.91	5.33	537	16.35	3.2
Naru	–	–	–	–	–	–
NeuroCard	1.03	1.51	5.09	158	71	9.9
UAE	1.02	1.48	4.35	151	72	9.9
FACE	**1.02**	**1.16**	**1.60**	**3.00**	10.74	1.2
(b) IMDB						
PG	2.92	47.3	2768	$1 \cdot 10^4$	**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	$1 \cdot 10^3$	0.32	–
lw-nn	1.23	8.89	35.0	405	0.63	4.7
lw-xgb	1.16	11.1	36.8	$1 \cdot 10^3$	0.3	0.99
MSCN	1.18	5.04	64.0	$2 \cdot 10^3$	0.85	4.2
FCN	1.09	2.58	8.49	119	3.13	1.5
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
UAE	1.02	1.50	2.64	14.3	62.5	6.2
FACE	**1.02**	**1.21**	**1.54**	**2.85**	11.6	1.2
(c) BJAQ						
PG	1.46	9.94	30.4	$1 \cdot 10^3$	**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
FCN	1.06	1.94	5.42	51	3.05	1.5
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	–	–	–	–	–	–
UAE	1.03	1.24	1.51	7.65	12.7	9.2
FACE	**1.03**	**1.16**	**1.30**	**2.55**	11.8	0.37
(d) TPC-H						
PG	1.32	97.2	216	609	**0.26**	**0.01**
Sample	1.04	92	183	571	1.67	–

Table 2 continued

Estimator	50th	95th	99th	Max	Latency	Model size
MHIST	1.05	3.81	5.50	56.1	365	5.6
KDE	1.05	2.46	5.38	40.5	0.48	–
lw-nn	1.09	3.18	5.32	25.9	0.85	0.52
lw-xgb	1.06	3.21	4.00	21.2	0.42	0.81
MSCN	1.06	3.99	10.7	445	0.93	0.25
FCN	1.06	1.74	3.18	20.0	2.91	1.5
DeepDB	1.04	1.46	2.13	9.50	6.35	0.45
Naru	1.04	1.42	2.09	10.5	8.89	9.80
NeuroCard	–	–	–	–	–	–
UAE	1.04	1.37	1.81	8.49	8.85	9.80
FACE	**1.03**	**1.17**	**1.41**	**1.74**	7.90	0.22

Bold values for accuracy indicate the most accurate results achieved. For latency and model size, bold values represent the smallest latency and model sizes, respectively

(1) PG [41]: Postgres, using independent histograms.
(2) Sample [29, 58]: the method sampled a number of records to do CE. The sampled size was set to 1% of each dataset.
(3) MHIST [40]: the method stored all entries in the PDF using a compression technique.
(4) KDE [16, 22]: used kernel density estimation for CE.
(5) lw-nn [7]: a query-driven method that trained a neural network to estimate the cardinality.
(6) lw-xgb [7]: a query-driven method that trained a gradient boost tree to estimate the cardinality.
(7) MSCN [25]: a query-driven method that used multi-set convolutional network.
(8) FCN [23]: a query-driven method that used a fully connected network and pooling layer to estimate the cardinality.
(9) DeepDB [17]: the method used sum-product network.
(10) Naru [56]: the method used the autoregressive model.
(11) NeuroCard [55]: the method extended Naru to support multi-table. It could handle large domain size data by splitting columns.
(12) UAE [54]: the method that extended the autoregressive methods, *i.e.,* Naru and NeuroCard, to incorporate query-driven training.

We obtained codes of baselines from the authors and an experimental work [53]. 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 [56]. 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 Sect. 9.4. (3) We randomly selected a tuple from the table and used the attributes of the tuple as the literals. Since the selected tuple always satisfies all the predicates in the query, the generated queries have a minimum cardinality of 1. 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 [26]. It was defined as $Q\text{-error} = \max\{\frac{car(\theta)}{\widehat{car(\theta)}}, \frac{\widehat{car(\theta)}}{car(\theta)}\}$, 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 Intel(R) Xeon(R) Silver 4110 CPU, a Nvidia 2080ti GPU and 128GB RAM.

9.2 Overall evaluation

9.2.1 Comparison of accuracy

Table 2 shows the Q-errors of different CE algorithms. Methods are grouped as traditional, query-driven and data-

driven, respectively. The results could be ranked as FACE > Naru/NeuroCard/UAE > DeepDB >> FCN > 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. In particular, 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 [56], 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, because our framework could well model the joint distribution of data with different types.

FACE performed better than Naru, NeuroCard and UAE. 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. UAE extended Naru and NeuroCard to incorporate query-driven training and had the same network structure as Naru and NeuroCard. Therefore, we used the same model parameters for UAE with Naru and NeuroCard for a fair comparison. On Power and IMDB, FACE outperformed both of NeuroCard and UAE 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 and UAE 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 and UAE 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 as it can address this problem through coupling layers in NF model.

FACE outperformed 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, FCN were 26.8, 8.10, 176, 7.66, 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 [46]. MHIST loses information because of the compression and KDE cannot handle multi-dimensional data well by kernel functions.

9.2.2 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 10 ms on 4 datasets) was applicable in practice. FACE was faster than Naru/NeuroCard/UAE, especially on large domain sizes. For example, on Power, FACE was 7 × faster than NeuroCard and UAE. 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.

9.2.3 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 [53]. 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.

9.3 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 [53] 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 Sect. 9.1.

9.3.1 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[5] and DeepDB, two representative data-driven methods. The results in Fig. 10a 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 has 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 to Naru and DeepDB. Moreover, FACE could perform well on large domain size data due to the inherent support of the NF model for modeling continuous data. This eliminates the need for approximate learned embeddings (Naru) or heuristic assumptions (DeepDB), enabling accurate modeling of large domain size data.

9.3.2 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. 10b 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 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.

9.4 Like predicates evaluation

We evaluated the queries with Like predicates generated in IMDB. Similar to Sect. 9.1, 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

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

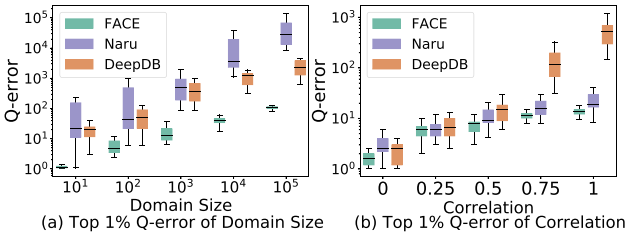

Fig. 10 Evaluation of synthetic datasets

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 [46] 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 (43.7MB) and FACE (67.8MB) 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.

9.5 Multi-table evaluation

In this section, we evaluated the CE methods on multiple tables using the widely used benchmark JOB-light [25]. The 6 tables used in JOB-light collectively occupy a total size of 2.7 GB. The results of different methods are shown in Table 4. The *Single-Model* and *Multi-Models* methods that are introduced in Sect. 8 were evaluated here, respectively, and were represented as FACE-Single and FACE-Multiple. We trained FACE-Single on 7 M samples from the full outer join table and trained FACE-Multiple on 2 M samples for each of the join templates.

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-

Table 3 Evaluation of `Like` predicates

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

Bold values for accuracy indicate the most accurate results achieved. For latency and model size, bold values represent the smallest latency and model sizes, respectively

Multiple, even if we trained a model for each join template, the model size was smaller than the baselines. The reasons were twofold. (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, NeuroCard and UAE, 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. We could see that in terms of accuracy, FACE-Multiple outperformed the FACE-Single and other baselines because it provided more fine-grained models for different templates. However, FACE-Multiple is less flexible compared with FACE-Single because it relies on historical queries to generate query templates.

We also reported the building time and the memory footprint for different methods on JOB-light, as shown in Fig. 11. In terms of the building time, besides the time of model training, for data-driven methods, it includes the time of sampling train data points. For query-driven methods, it includes the time to obtain labels (true cardinalities) for training queries.

Regarding the building time, Fig. 11a shows that FACE-Multiple (92 min) has a slightly longer training time compared to other data-driven methods such as FACE-Single (42 min) and NeuroCard (49 min) because it trains over multiple (27 for JOB-light) small models. However, it is still faster than query-driven methods like MSCN, which takes 182 min. This is because FACE-Multiple does not require obtaining the true labels (cardinalities) for training queries, which can be time-consuming for query-driven methods, especially in the case of multiple tables. Therefore, the building cost of FACE-Multiple is relatively small among all the methods.

Additionally, we have also evaluated the training and testing memory footprints. From Fig. 11b and c, we can observe that both FACE-Single and FACE-Multiple have small memory footprints compared with other methods, both for training and testing. This is attributed to the compact structures of the normalizing flow model used in FACE.

9.6 End-to-end evaluation

To evaluate the effectiveness of different cardinality estimators in practical query optimization, we injected their estimation results into PostgreSQL (PG), a widely used open-source database, and tested with JOB-light workload. To be specific, for each estimator to be compared, we allowed the query optimizer in PG to utilize the estimator's results to select the plan and execute the queries accordingly. This allowed us to evaluate the impact of different CE methods on the overall execution performance. The results are presented in Table 4.

In the table, on JOB-light, we first reported the estimation accuracy of different methods for all the sub-plan queries whose cardinalities have to be estimated in the planning phase for query optimization. Totally, we have 696 sub-plan queries, denoted by JOB-light-all. The column Exec.Time in Table 4 represented the total execution time of the selected plans, while the column Plan Time represented the planning time, including the time for plan selection plus the cardinality estimations. Furthermore, the column Improvement showed the improvement in total time compared to the baseline PG, *i.e.,* simply running JOB-light workload in PG without utilizing any additional cardinality estimator.

From Table 4, we could observe that the impact of different CE methods on the overall execution time of the complex JOB-light workload is negligible, accounting for less than 0.5%. In terms of overall execution efficiency, more accurate cardinality estimates of these sub-plan queries enabled the optimizer to select better execution plans, leading to a significant reduction in the execution time (up to 30 min).

For instance, FACE-Multiple only consumed 14 s while reducing the Exec.Time from 3.06 h in PG to 2.57 h, which is very close to the optimal execution time (2.55 h) achieved using true cardinalities (TrueCard). This brought a 16% improvement in the overall execution time compared to using PG alone because of these highly accurate cardinality estimates for the sub-plan queries. As shown in Table 4,

Table 4 End-to-end evaluation on JOB-light

Estimator	Workload JOB-light-all				JOB-light				Model size	End-to-end execution		Total time, h	Improvement, %
	50th	95th	99th	Max	50th	95th	99th	Max		Exec. time, h	Plan time, s		
PG	2.58	91.5	842	3e3	7.97	797	$3 \cdot 10^3$	$3 \cdot 10^3$	70 KB	3.06	**4**	3.06	0.0
TrueCard	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	–	**2.55**	**4**	**2.55**	**16.7**
MSCN	2.05	54.2	215	1e3	3.01	136	$1 \cdot 10^3$	$1 \cdot 10^3$	2.7 MB	2.98	**11**	2.98	2.7
E2E	2.34	78.9	243	1e3	3.51	139	244	272	–	3.02	12	3.02	1.2
FCN	1.49	5.62	42.7	227	1.33	6.76	21.2	30.5	7.7 MB	2.77	12	2.77	9.5
DeepDB	1.67	4.60	38.0	46.4	1.32	4.90	33.7	72.0	3.7 MB	2.75	33	2.76	9.8
UAE	1.94	20.3	50.9	140	1.56	6.29	17.2	33.0	4.1 MB	2.92	49	2.93	4.2
NeuroCard	2.15	17.2	36.9	117	1.57	5.91	8.48	8.51	3.8 MB	2.86	43	2.87	6.2
FACE-Single	1.23	4.50	11.7	28.8	1.22	4.84	8.03	8.25	10.5 MB	2.67	47	2.68	12.4
FACE-Multiple	**1.04**	**3.24**	**9.72**	**18.9**	**1.15**	**4.49**	**7.41**	**7.83**	**1.73 MB**	**2.57**	14	**2.57**	**16.0**

Bold values for accuracy highlight the most accurate outcomes. Bold values for time and model size indicate the shortest time and smallest model size achieved. Additionally, bold values for improvement denote the highest degree of improvement

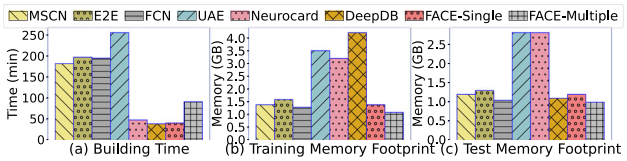

Fig. 11 Building time and memory footprint on JOB-light

FACE-Multiple achieved a median q-error of only 1.04 on JOB-light-all, significantly outperforming other methods such as DeepDB (1.67) and FCN (1.49).

9.7 Case study

In this section, we present a case study focusing on the 60-th query of JOB-light. Figure 12 illustrates the execution plans selected based on several representative methods, including TrueCard, FACE-single, FACE-multiple, NeuroCard, DeepDB, and PG, which shows the estimated cardinalities of these methods for each sub-plan node, as well as the actual execution time of the plan. Since FACE-single and FACE-multiple lead to the same plan, we present them together in one graph referred as FACE and report the estimated cardinality of FACE-multiple.

We can observe that FACE enables the optimizer to select an optimal plan identical to TrueCard, resulting in a significantly reduced query execution time because of the accurate estimated cardinality. On the other hand, NeuroCard, due to its underestimation of the join result between table ci and t (q-error of 10.8), failed to identify the optimal join order. In addition, both DeepDB and PG exhibit large estimation errors, with the maximum q-errors reaching 39.6 and 1058, respectively. As a consequence, the misguided optimizer selected poor plans, leading to longer execution times compared to the FACE plan.

Overall, these findings highlight the capabilities of FACE in accurately estimating cardinalities, facilitating the selection of better plans, thus achieving significantly shorter query execution times.

9.8 Parallel queries evaluation

In this section, we evaluated the performance of different methods in the case of parallel queries. We compared our FACE-CAGroup algorithm with six baselines: ApproxLabel, FCN-batch, UAE-parallel, FACE, FACE-Greedy and FACE-IG. ApproxLabel [8] refers to a sampling approach aiming at approximating cardinality while minimizing the number of sample points. FCN-batch is a newly added baseline (the batch version of FCN) that we take the encodings of all parallel queries into a batch and send to the FCN [23] model for estimation in parallel. UAE-parallel is another newly added baseline that refers to

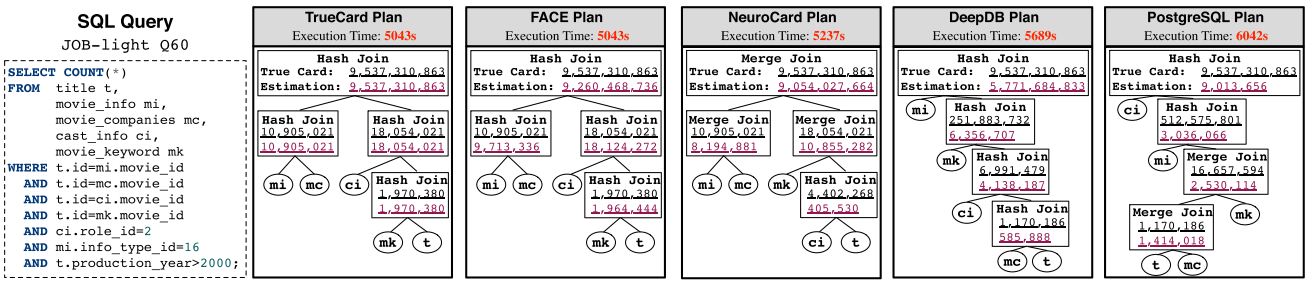

Fig. 12 A case study of JOB-light Q60

the multi-process version of UAE [54]. Specifically, we use 4 processes. FACE is the one in Sect. 9.2 without parallel data sharing. FACE-Greedy, FACE-IG and FACE-CAGroup first group parallel queries using Greedy, IG and CAGroup algorithms, respectively. After that, they use PCEA to generate data points shared by queries within each group to perform CE.

We evaluated these 7 methods on 4 datasets: Power, IMDB, BJAQ and TPC-H. For each dataset, all the methods used the same models in Sect. 9.2. The CE accuracy and average latency of different methods are shown in Table 5. The 2000 testing queries were generated based on the same method in Sect. 9.1. We first assume that all the generated queries come in parallel and then vary the number of parallel queries in Sect. 8.7.3.

For the methods with parallel data points sharing, we set the query similarity threshold ϵ as 0.5 and set the number of sampled data points b for each group as $\frac{K}{\epsilon}$, because ϵ approximates the proportion of the b data points that each query within the group can use. Therefore, taking $b = \frac{K}{\epsilon}$ allows each query in the group to approximately have at least K available data points. For all the methods, we set the number of buckets $|B| = 100$ and adaptively sample until converge.

9.8.1 Comparison of latency

Table 5 shows the average latency of different methods. The results were ranked as FCN-batch < FACE-CAGroup < ApproxLabel < FACE-Greedy < FACE < FACE-IG < UAE-parallel. For example, on dataset Power, FACE-CAGroup (0.91 ms) was 436 times faster than FACE-IG (397 ms), 79 times faster than UAE-parallel (71.7 ms), 12 times faster than FACE (10.74 ms), and 5 times faster than FACE-Greedy (5.03 ms).

ApproxLabel was relatively slower than FACE-CAGroup because ApproxLabel, as a sampling-based method, needs to execute the query on sample. Although FCN-batch was faster than FACE-CAGroup, its accuracy was much worse because FCN-batch is a query-driven method that relies on the consistence of training and test workload, which is not generalizable enough. UAE-parallel was much slower than FACE-CAGroup, because FACE-CAGroup reduces the amount of computation for similar parallel queries by reducing the totally number of sampled points. FACE was slow because supposing that every single query has to sample K data points for MC integration, n queries lead to nK data points. This would yield a large number of computations that are hard to process simultaneously by GPU with limited computation capacity. As a result, these computations had to be splitted into batches and computed by GPU batch by batch, which resulted in high latency. FACE-CAGroup and FACE-Greedy were faster than FACE because they reduced the total number of sampled data points, thus reducing the total number of computations. Besides, FACE-CAGroup was faster than FACE-Greedy because FACE-CAGroup divided the queries into fewer groups. FACE-IG was inefficient, because although FACE-IG could divide the queries into few groups, it has to perform Greedy many times, which was time-consuming. FACE-CAGroup was much faster than UAE-parallel, because FACE-CAGroup reduces the amount of computation for similar parallel queries by reducing the totally number of sampled points.

Although FACE-CAGroup greatly reduced the estimation latency, the accuracy did not decrease much, which we will analyze next.

9.8.2 Comparison of accuracy

We reported the Q-errors on all the testing queries in Table 5. We could observe from the table that all the methods performed similarly on the entire distribution of Q-error for all datasets. For example, the median (Max-quantile) Q-errors on the dataset BJAQ for ApproxLabel, FCN-batch, UAE-parallel, FACE, FACE-Greedy, FACE-IG and FACE-CAGroup were 1.05 (184), 1.12 (77.4), 1.03 (7.65), 1.03 (2.55), 1.03 (3.05), 1.03 (4.26) and 1.03 (2.86), respectively. This demonstrated that the methods using parallel data sharing obtained comparable CE accuracy compared to FACE. The reason is that queries in each group are similar, and thus the data points can be well shared. Therefore,

Table 5 Q-errors and latency (ms) on 4 datasets for parallel queries

Dataset	Method	50th	95th	99th	MAX	Latency
Power	ApproxLabel	1.14	2.16	103	419	1.34
	FCN-batch	1.07	4.70	26.8	455	**0.39**
	UAE-parallel	**1.02**	1.48	4.35	151	71.7
	FACE	**1.02**	**1.16**	**1.60**	**3.00**	10.74
	FACE-Greedy	**1.02**	1.34	1.95	4.30	5.03
	FACE-IG	**1.02**	1.37	2.00	5.32	397
	FACE-CAGroup	**1.02**	1.34	2.00	4.80	0.91
IMDB	ApproxLabel	1.08	1.96	5.51	291	1.47
	FCN-batch	1.23	8.89	35.0	405	**0.34**
	UAE-parallel	**1.02**	1.50	2.64	14.3	62.2
	FACE	**1.02**	**1.21**	**1.54**	**2.85**	11.6
	FACE-Greedy	1.03	1.67	2.79	3.22	4.57
	FACE-IG	1.03	1.72	3.01	3.92	401
	FACE-CAGroup	1.03	1.59	2.85	3.46	1.31
BJAQ	ApproxLabel	1.05	1.49	2.87	184	1.53
	FCN-batch	1.12	5.46	14.1	77.4	**0.37**
	UAE-parallel	**1.03**	1.24	1.51	7.65	12.5
	FACE	**1.03**	**1.16**	**1.3**	**2.55**	11.8
	FACE-Greedy	**1.03**	**1.16**	1.42	3.05	5.58
	FACE-IG	**1.03**	1.17	1.49	4.26	382
	FACE-CAGroup	**1.03**	**1.16**	1.41	2.86	0.98
TPC-H	ApproxLabel	1.04	89	157	482	2.87
	FCN-batch	1.06	1.74	3.18	20.0	**0.40**
	UAE-parallel	1.04	1.37	1.81	8.49	9.55
	FACE	**1.03**	**1.17**	**1.41**	**1.74**	7.90
	FACE-Greedy	**1.03**	1.19	1.49	2.77	3.81
	FACE-IG	**1.03**	1.19	1.62	2.48	281
	FACE-CAGroup	**1.03**	1.18	1.59	2.03	0.74

Bold values for accuracy indicate the most accurate results achieved. For latency and model size, bold values represent the smallest latency and model sizes, respectively

through parallel data sharing, each query can still get enough data points to obtain high CE accuracy.

To summarize, FACE-CAGroup achieved the state-of-the-art accuracy with an order of magnitude improvement in efficiency over FACE for parallel queries.

9.8.3 Evaluation of hyper-parameters

In this section, we evaluated how the performance of FACE-CAGroup would be affected by two important factors, i.e., the number of parallel queries n and the similarity threshold ϵ.

Evaluation of n. We varied the number of parallel queries n from 20 to 2000 for the 2000 queries used for Power in Sect. 9.8. We reported the average latency of FACE-CAGroup with $\epsilon = 0.5$ in Fig. 13.

For each batch of parallel queries, FACE-CAGroup first divides them into several groups by CAGroup. Then it uses PCEA to iteratively generate data points for each group to perform CE. Between batches, the sampled data sharing in Sect. 6.2 are also used to reuse the data points of previous batches.

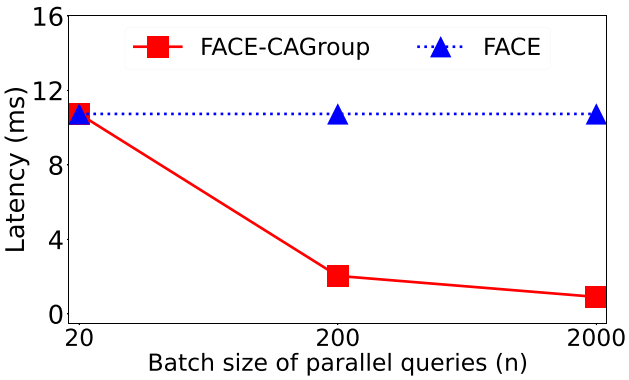

Fig. 13 Evaluation of number of parallel queries on Power

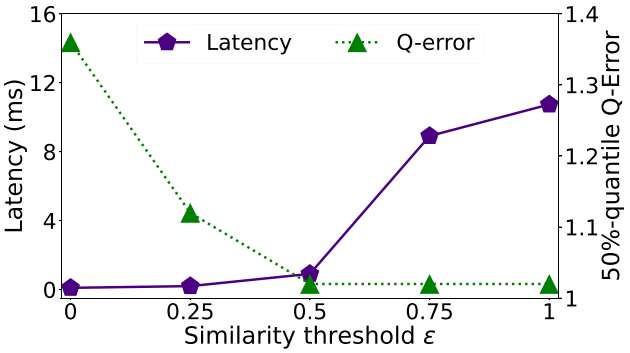

Fig. 14 Evaluation of similarity threshold ϵ on `Power`

From Fig. 13, we could observe that when $n = 20$, `FACE-CAGroup` had the same average latency (10.74 ms) as `FACE`. This is because the computations of `FACE` incurred by nK sampled data points do not exceed the computation capacity of the GPU until $n = 20$. When $n > 20$, the average latency of `FACE` will remain the same because the overall computation exceeded the computation capacity of GPU and had to be computed batch by batch. However, we could observe from Fig. 13 that along with the n increasing, the average latency of `FACE-CAGroup` decreased from 10.74 to 0.91 ms. This is because `FACE-CAGroup` reduced the total number of data points by grouping the n queries and sharing sampling points within each group. For the same 2000 queries, a larger n leads to a smaller total number of groups, *i.e.,* less data points, and is therefore more efficient. On the other hand, along with the n increasing, the CE accuracy does not change. The reason is that each query is estimated by sharing the b sampled data points of its group, and b does not change with n.

Evaluation of ϵ. We varied the similarity threshold ϵ from 0 to 1 with $n = 2000$ on `Power`. Figure 14 shows the average latency and the 50% quantile Q-error of `FACE-CAGroup`. We take b as $\frac{K}{\epsilon}$ for $\epsilon \neq 0$ and take b as $10K$ for $\epsilon = 0$. We could observe that the latency increased from 0.28 to 10.74 ms along with the ϵ increasing. This is because larger ϵ yields more groups and more sampled data points, thus incurring higher latency. In addition, with the ϵ increasing, the Q-error decreased from 1.36 to 1.02 and then remained stable. The reason is that the queries in each group generated by larger ϵ are more similar. Therefore, data points can be better shared in each group, which leads to more accurate CE. When ϵ reached a relatively large value, *e.g.,* 0.5, the queries within each group can get enough number of data points to obtain accurate CE. After that, increasing ϵ brings almost no further accuracy improvement. Thus, we set ϵ to 0.5 as in Sect. 9.8, so as to obtain a good trade-off between latency and accuracy.

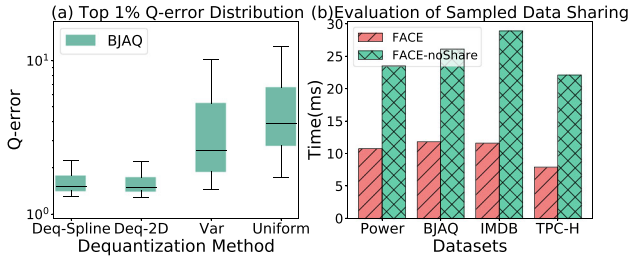

Fig. 15 Variance evaluation

9.9 Variance evaluation

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

9.9.1 Dequantization

We compared our spline dequantization (`Deq-Spline`, proposed in Sect. 4) with three baselines: uniform dequantization [51], variational dequantization [18] and 2-dimensional continuity dequantization. The first one (`Uniform`) is discussed in Sect. 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 to build a continuous PDF on 2-dimensional data.

As shown in Fig. 15a, 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.

9.9.2 Sampled data sharing

We evaluated the sampled data sharing proposed in Sect. 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` and `FACE` on 3 real-world datasets. As shown in Fig. 15b, we improved the efficiency two times because `FACE` shared the sampled data with similar queries, and thus the convergence was fast.

9.10 Data updates evaluation

We studied the impact of data updates on `FACE`. Following [56], 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

Table 6 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

of `NoModelUpdate` denoted that we trained on current arrived data and directly estimated the cardinality of the workload when 20% data were added, without any model update. The row of `Inc-Training` denoted that when the 20% new data were 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 6, 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.

10 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 propose a Monte Carlo method to estimate the cardinality based on the `FACE` model and propose a grouping technique to process parallel queries. Besides, we discuss how to support join queries. The results show that our method gains 50× performance improvement on accuracy.

Acknowledgements We would like to thank the anonymous reviewers for their precious comments. This work is supported by NSF of China (61925205, 62232009, 62102215), Huawei, TAL education, and Zhongguancun Lab.

References

1. Beliakov, G.: Monotonicity preserving approximation of multivariate scattered data. BIT Numer. Math. **45**, 653–677 (2005)
2. Blanchette, M., Kim, E., Vetta, A.: Clique cover on sparse networks. In: Bader, D.A., Mutzel, P. (eds.) ALENEX, pp. 93–102. SIAM (2012)
3. Cerioli, M.R., Faria, L., Ferreira, T.O., Martinhon, C.A.J., Protti, F., Reed, B.A.: Partition into cliques for cubic graphs: planar case, complexity and approximation. Discret. Appl. Math. **156**(12), 2270–2278 (2008)
4. Chalupa, D.: On the efficiency of an order-based representation in the clique covering problem. In: Soule, T., Moore, J.H. (eds.) GECCO, pp. 353–360. ACM (2012)
5. Chalupa, D.: Construction of near-optimal vertex clique covering for real-world networks. Comput. Inf. **34**(6), 1397–1417 (2015)
6. Dinh, L., Krueger, D., Bengio, Y.: NICE: non-linear independent components estimation. In: Y. Bengio and Y. LeCun, (eds.) ICLR (2015)
7. Dutt, A., Wang, C., Nazi, A., Kandula, S., Narasayya, V., Chaudhuri, S.: Selectivity estimation for range predicates using lightweight models. Proc. VLDB Endow. **12**(9), 1044–1057 (2019)
8. Dutt, A., Wang, C., Nazi, A., Kandula, S., Narasayya, V.R., Chaudhuri, S.: Selectivity estimation for range predicates using lightweight models. Proc. VLDB Endow. **12**(9), 1044–1057 (2019)
9. Fritsch, F.N., Carlson, R.E.: Monotone piecewise cubic interpolation. SIAM J. Numer. Anal. **17**(2), 238–246 (1980)
10. Garey, M.R., Johnson, D.S.: Computers and intractability: a guide to the theory of NP-completeness. W. H. Freeman (1979)
11. Germain, M., Gregor, K., Murray, I., Larochelle, H.: MADE: masked autoencoder for distribution estimation. In: ICML, vol. 37, pp. 881–889 (2015)
12. Gharibshah, Z., Zhu, X., Hainline, A., Conway, M.: Deep learning for user interest and response prediction in online display advertising. Data Sci. Eng. **5**(1), 12–26 (2020)
13. Goodfellow, I., Pouget-Abadie, E.A.: Generative adversarial nets. NIPS **27**, 2672–2680 (2014)
14. Han, L., Schumaker, L.L.: Fitting monotone surfaces to scattered data using c1 piecewise cubics. SIAM J. Numer. Anal. **34**(2), 569–585 (1997)
15. Hasan, S., Thirumuruganathan, S., Augustine, J., Koudas, N., Das, G.: Deep learning models for selectivity estimation of multi-attribute queries. In: SIGMOD, pp. 1035–1050. ACM (2020)
16. Heimel, M., Kiefer, M., Markl, V.: Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In: SIGMOD, pp. 1477–1492. ACM, (2015)
17. Hilprecht, B., Schmidt, A., Kulessa, M., Molina, A., Kersting, K., Binnig, C.: Deepdb: learn from data, not from queries! VLDB **13**(7), 992–1005 (2020)
18. Ho, J., Chen, X., Srinivas, A., Duan, Y., Abbeel, P.: Flow++: Improving flow-based generative models with variational dequantization and architecture design. In: ICML, vol. 97, pp. 2722–2730. PMLR (2019)
19. Hoogeboom, E., Cohen, T.S., Tomczak, J.M.: Learning discrete distributions by dequantization. arXiv preprint arXiv:2001.11235 (2020)
20. I. household electric power consumption data set. https://github.com/gpapamak/maf, (2021)
21. Karp, R.M.: Reducibility among combinatorial problems. In: Miller, R.E., Thatcher, J.W. (eds.) The IBM research symposia series, pp. 85–103. Plenum Press, New York (1972)
22. Kiefer, M., Heimel, M., Breß, S., Markl, V.: Estimating join selectivities using bandwidth-optimized kernel density models. VLDB **10**(13), 2085–2096 (2017)
23. Kim, K., Jung, J., Seo, I., Han, W., Choi, K., Chong, J.: Learned cardinality estimation: an in-depth study. In SIGMOD, pp. 1214–1227. ACM, (2022)
24. Kingma, D.P., Welling, M.: Auto-encoding variational bayes. In: ICLR, (2014)
25. Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P.A., Kemper, A.: Cardinalities: Estimating correlated joins with deep learning. In: CIDR. www.cidrdb.org (2019)
26. Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P.A., Kemper, P.A.: Learned cardinalities: estimating correlated joins with deep learning. In: CIDR (2019)

27. Kobyzev, I., Prince, S., Brubaker, S.: Normalizing flows: an introduction and review of current methods. IEEE Trans. Pattern Anal. Mach. Intell. **43**, 3964–3979 (2020)
28. Leis, V., Gubichev, A., Mirchev, A., Boncz, P.A., Kemper, A., Neumann, T.: How good are query optimizers, really? VLDB **9**(3), 204–215 (2015)
29. Leis, V., Radke, B., Gubichev, A., Kemper, A., Neumann, T.: Cardinality estimation done right: Index-based join sampling. In: CIDR. www.cidrdb.org (2017)
30. Lepage, G.P.: Adaptive multidimensional integration: Vegas enhanced. J. Comput. Phys. **439**, 110386 (2021)
31. Li, G., Zhou, X., Cao, : AI meets database: AI4DB and DB4AI. In: SIGMOD, pp. 2859–2866 (2021)
32. Li, G., Zhou, X., Cao, L.: Machine learning for databases. Proc. VLDB Endow. **14**(12), 3190–3193 (2021)
33. Li, G., Zhou, X., Chai, C.: AI meets database: a survey. In: TKDE (2021)
34. Li, G., Zhou, X., Sun, J., Yu, X., Han, Y., Jin, L., Li, W., Wang, T., Li, S.: opengauss: An autonomous database system. Proc. VLDB Endow. **14**(12), 3028–3041 (2021)
35. Li, M., Wang, H., Li, J.: Mining conditional functional dependency rules on big data. Big Data Min. Anal. **03**(01), 68 (2020)
36. Müller, T., McWilliams, B., Rousselle, F., Gross, M., Novák, J.: Neural importance sampling. ACM Trans. Graph. **38**(5), 1–19 (2019)
37. Ortiz, J., Balazinska, M., Gehrke, J., Keerthi, S.S.: An empirical analysis of deep learning for cardinality estimation. arXiv preprint arXiv:1905.06425 (2019)
38. Papamakarios, G., Nalisnick, E., Rezende, D.J., Mohamed, S., Lakshminarayanan, B.: Normalizing flows for probabilistic modeling and inference. J. Mach. Learn. Res. **22**(57), 1–64 (2021)
39. Lepage, G.P.: A new algorithm for adaptive multidimensional integration. J. Comput. Phys. **27**(2), 192–203 (1978)
40. Poosala, V., Ioannidis, Y.E., Haas, P.J., Shekita, E.J.: Improved histograms for selectivity estimation of range predicates. In: SIGMOD, pp. 294–305. ACM Press (1996)
41. PostgreSQL. https://www.postgresql.org/ (2021). Accessed 14 Sep 2021
42. Rezende, D.J., Mohamed, S.: Variational inference with normalizing flows. In: ICML, vol. 37, pp. 1530–1538 (2015)
43. Selinger, P.G., Astrahan, M.M., Chamberlin, D.D., Lorie, R.A., Price, T.G.: Access path selection in a relational database management system. In: Bernstein, P.A. (ed), SIGMOD, pp. 23–34. ACM (1979)
44. Set, B.M.-S.A.-Q.D.D. https://archive.ics.uci.edu/ml/datasets/Beijing+Multi-Site+Air-Quality+Data (2021). Accessed 14 Sep 2021
45. Strash, D., Thompson, L.: Effective data reduction for the vertex clique cover problem. In: Phillips, C.A. and Speckmann, B. (eds), ALENEX, pp. 41–53. SIAM (2022)
46. Sun, J., Li, G.: An end-to-end learning-based cost estimator. VLDB **13**(3), 307–319 (2019)
47. Sun, J., Li, G., Tang, N.: Learned cardinality estimation for similarity queries. In: SIGMOD, pp. 1745–1757 (2021)
48. Sun, J., Zhang, J., Sun, Z., Li, G., Tang, N.: Learned cardinality estimation: a design space exploration and a comparative evaluation. VLDB, (2021)
49. Theis, L., van den Oord, A., Bethge, M.: A note on the evaluation of generative models. In: Bengio, Y. and LeCun, Y. (eds.), ICLR, (2016)
50. Tian, S., Mo, S., Wang, L., Peng, Z.: Deep reinforcement learning-based approach to tackle topic-aware influence maximization. Data Sci. Eng. **5**(1), 1–11 (2020)
51. Uria, B., Murray, I., Larochelle, H.: RNADE: the real-valued neural autoregressive density-estimator. In: Burges, C.J.C., Bottou, L., Ghahramani, Z., and Weinberger, K.Q. (eds.) NIPS, pp. 2175–2183 (2013)
52. Wang, J., Chai, C., Liu, J., Li, G.: FACE: a normalizing flow based cardinality estimator. Proc. VLDB Endow. **15**(1), 72–84 (2021). https://doi.org/10.14778/3485450.3485458
53. Wang, X., Qu, C., Wu, W., Wang, J., Zhou, Q.: Are we ready for learned cardinality estimation? Proc. VLDB Endow. **14**(9), 1640–1654 (2021)
54. Wu, P., Cong, G.: A unified deep model of learning from both data and queries for cardinality estimation. In: SIGMOD, pp. 2009–2022. ACM (2021)
55. Yang, Z., Kamsetty, A., Luan, S., Liang, E., Duan, Y., Chen, X., Stoica, I.: Neurocard: one cardinality estimator for all tables. Proc. VLDB Endow. **14**(1), 61–73 (2020)
56. Yang, Z., Liang, E., Kamsetty, A., Wu, C., Duan, Y., Chen, P., Abbeel, P., Hellerstein, J.M., Krishnan, S., Stoica, I.: Deep unsupervised cardinality estimation. VLDB **13**(3), 279–292 (2019)
57. Yu, X., Li, G., Chai, C., Tang, N.: Reinforcement learning with tree-lstm for join order selection. In: ICDE, pp. 1297–1308. IEEE (2020)
58. Zhao, Z., Christensen, R., Li, F., Hu, X., Yi, K.: Random sampling over joins revisited. In: SIGMOD 2018, pp. 1525–1539. ACM (2018)
59. Zhou, X., Sun, J., Li, G., Feng, J.: Query performance prediction for concurrent queries using graph embedding. Proc. VLDB Endow. **13**(9), 1416–1428 (2020)
60. Zhu, R., Wu, Z., Han, Y., Zeng, K., Pfadler, A., Qian, Z., Zhou, J., Cui, B.: FLAT: fast, lightweight and accurate method for cardinality estimation. VLDB **14**(9), 1489–1502 (2021)
61. Ziegler, Z.M., Rush, A.M.: Latent normalizing flows for discrete sequences. In: Chaudhuri, K. and Salakhutdinov, R. (eds.) ICML, vol. 97, pp. 7673–7682. PMLR (2019)
62. Zuckerman, D.: Linear degree extractors and the inapproximability of max clique and chromatic number. In: Kleinberg, J.M. (ed.) STOC, pp. 681–690. ACM (2006)