AutoCE: An Accurate and Efficient Model Advisor for Learned Cardinality Estimation

Jintao Zhang[1], Chao Zhang[1], Guoliang Li[1], Chengliang Chai[2]
[1]*Department of Computer Science, Tsinghua University,* [2]*School of Computer Science, Beijing Institute of Technology*
{zjt21@mails., cycchao@mail., liguoliang@}tsinghua.edu.cn, ccl@bit.edu.cn

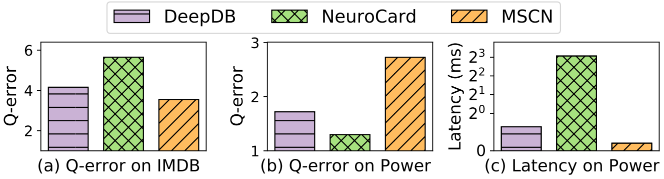
***Abstract*—Cardinality estimation (CE) plays a crucial role in many database-related tasks such as query generation, cost estimation, and join ordering. Lately, we have witnessed the emergence of numerous learned CE models. However, no single CE model is invincible when it comes to the datasets with various data distributions. To facilitate data-intensive applications with accurate and efficient cardinality estimation, it is important to have an approach that can judiciously and efficiently select the most suitable CE model for an arbitrary dataset.**

In this paper, we study a new problem of selecting the best CE models for a variety of datasets. This problem is rather challenging as it is hard to capture the relationship from various datasets to the performance of disparate models. To address this problem, we propose a model advisor, named `AutoCE`, which can adaptively select the best model for a dataset. The main contribution of `AutoCE` is the learning-based model selection, where deep metric learning is used to learn a recommendation model and incremental learning is proposed to reduce the labeling overhead and improve the model robustness. We have integrated `AutoCE` into PostgreSQL and evaluated its impact on query optimization. The results showed that `AutoCE` achieved the best performance (27% better) and outperformed the baselines concerning accuracy (2.1x better) and efficacy (4.2x better).

I. Introduction

Machine learning (ML) based CE models [4], [8], [11], [14]–[17], [20], [28], [30], [33], [35], [37], [38], [45] have recently attracted significant attention because it can harness the strong learning and representation ability of ML models to achieve superior performance [7], [10], [29], [32]. There are various types of learned CE models that either (1) encode the query workload to model the relationship between queries and their cardinalities (*query-driven* approaches [4], [11], [28]), or (2) encode the datasets to model the joint data distribution (*data-driven* approaches [8], [30], [35], [37], [38]).

Example 1 (Motivation): Figure 1 shows an experimental study of several typical learned CE models on different datasets. Figure 1(a) reports the estimation accuracy of 3 methods (DeepDB [8], NeuroCard [37] and MSCN [11] on `IMDB` dataset. Figure 1(b) reports the accuracy of these methods on a different `Power` dataset. Figure 1(c) reports the inference efficiency of the CE models.



(a) Q-error on IMDB (b) Q-error on Power (c) Latency on Power

Fig. 1. Experiment of CE models over different datasets.

Given the results, we have the first observation that on dataset `IMDB`, the accuracy is MSCN > DeepDB > Neurocard, while on `Power`, it is Neurocard > DeepDB > MSCN. The estimation accuracy varies because datasets have diverse and complicated data characteristics [29], [32], e.g., the number of tables, the number of joins, data correlation and skewness. There is no single CE model to perform well across the entire feature space. For example, MSCN performs better on `IMDB` mainly because the dataset contains multiple tables and it is hard for the other two data-driven methods to model the joint distribution across tables. Although we can train different CE models, test the performance and select the best one, it is rather time-consuming [3], which is not practical when datasets are diverse and evolving. Hence, given a dataset, it is necessary to develop a model advisor that can well capture the various data features and identify the best CE model precisely and quickly. In terms of the inference efficiency, MSCN performs the best, followed by DeepDB, and Neurocard is the most inefficient one. Given the `Power` dataset, if a user aims at generating millions of benchmarking queries with cardinality constraints [44], [46]–[48], the CE step of the generator needs to be efficient, so she is likely to choose MSCN. If she expects a good trade-off, DeepDB may be chosen rather than the most accurate Neurocard.

Applications. Overall, in this paper, we aim to build an intelligent CE model advisor that *given any particular dataset, it can efficiently choose the most suitable CE model customized to the user's requirements, e.g., the estimation accuracy and efficiency.* Many applications can benefit from the model advisor. A representative is the cloud data services [2], [13], which provide elastic data storage and querying services for multiple tenants. Using the model advisor, the cloud vendors can select an accurate CE model for an arbitrary dataset without the costly online learning. Particularly, the model advisor can rapidly select a new model when any data drift is detected [24]. Moreover, the model advisor can facilitate many applications that consider both accuracy and efficiency of CE models such as fraud detection [19] and query generation [48].

Challenges. An ideal solution is to take the given dataset and the user's requirement as input, then directly predict the

Guoliang Li and Chao Zhang are the corresponding authors. This work is supported by NSF of China (62232009, 61925205, 62102215, 62072261), Huawei, TAL education, and Beijing National Research Center for Information Science and Technology.

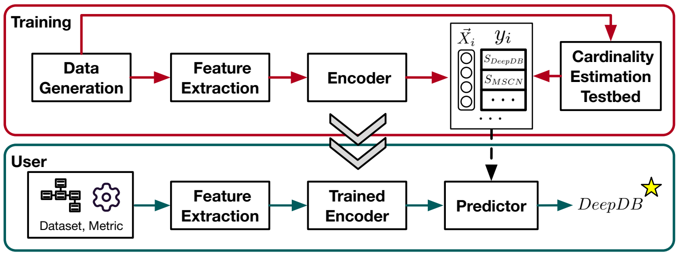


Fig. 2. An example of AutoCE: after the offline training, it selects a tailored cardinality estimation model with an arbitrary dataset and specified metrics.

best CE model among all candidates. There are two main challenges for learning such an advisor. (C1) the feature space of real-world datasets is huge, thus it is challenging to well capture the relationship from various datasets to the performance of different CE models. (C2) labeling the datasets is prohibitively expensive as it requires training and testing all the candidate CE models against each dataset to get the performance as the label.

To address the above challenges, we propose AutoCE, a model advisor for learned cardinality estimation. To well capture the relationship from data features to the CE models' performance, we model the features of a dataset as a graph and leverage deep metric learning to train a similarity-aware encoder of datasets based on the performance of CE models (addressing C1). To reduce the labeling overhead, we collect the samples that are not well predicted in the validation phase, then augment those samples with well-predicted samples by combining their features and labels (addressing C2).

To summarize, we have made the following contributions:

1) We study a new problem of selecting the learned CE models for a set of datasets. We propose a model advisor, AutoCE, which takes any dataset and metrics as input, then quickly selects a tailored CE model.
2) We propose a new learning method to encode the datasets as feature graphs, and leverage the deep metric learning to train a similarity-aware graph encoder.
3) We develop an incremental learning phase that identifies poorly-predicted samples and synthesizes new training samples to train the model incrementally.
4) We have integrated AutoCE into PostgreSQL v13.1 by injecting the estimated cardinalities into its query optimizer. The experimental results showed that AutoCE significantly improved the query performance by 27%, and improved the accuracy and efficacy by 2.1x and 4.2x, respectively.

II. Problem Statement

CE-model selection problem. Given a set of learned cardinality estimation (CE) models $\mathbb{M} = \{M_1, ., M_m\}$, and a set of datasets $\mathbb{D} = \{D_1, ., D_n\}$, the CE-model selection problem aims to select an optimal CE model $M_i \in \mathbb{M}$ for each dataset $D_j \in \mathbb{D}$ based on the specified performance metrics. The widely-used metrics include:

[1. *Q-error*] is an accuracy metric [22] for evaluating a selected model, defined as $Q\text{-}error = \frac{max(\widehat{card},card)}{min(\widehat{card},card)}$, where $\widehat{card}$ is estimated cardinality of a query and $card$ is the ground truth.

[2. *Inference latency*] is the efficiency metric for evaluating the inference overhead of a selected model, which is the running time of using the model for cardinality estimation.

[3. *E2E latency*] is used to denote the end-to-end (E2E) latency of answering a query using the selected CE model.

We quantify the Q-error/inference latency/E2E latency of a model M_i on a dataset D_j with three steps. First, we execute the given testing queries to get the true cardinalities (if not given, we generate the testing queries against D_j). Second, we use model M_i to estimate the cardinalities for the testing queries, then obtain the Q-error and inference latency. Third, we inject the estimated cardinalities to the database, then measure the E2E latency.

Example 2 (A Working Example of AutoCE): As shown in Figure 2, AutoCE consists of an offline training phase and an online prediction phase (i.e., without online learning). In the training phase, it generates a variety of datasets, extracts the dataset features, and learns an encoder to fit the relation from the data features to the performance of CE models. Particularly, the CE models are evaluated using the developed CE testbed, and the encoder is trained with different combinations of performance metrics. In the prediction phase, it takes as input any dataset and metric weights (e.g., [0.5, 0.5] represents 50% Q-error and 50% inference latency), then extracts its features and obtains the embedding using the trained encoder, and finally uses a predictor to select the best CE model.

III. Overview of AutoCE

An overview of AutoCE is presented in Figure 3, which is divided into four stages. Stage 1 is for data preparation (❶-❻). The training phase consists of stages 2 and 3 by walking through steps ❶-⓬. Particularly, stage 2 performs the deep metric learning to train a graph encoder, and stage 3 conducts the incremental learning based on the trained graph encoder and the original training data. Finally, stage 4 makes the recommendation with steps ❶-❼ .

Stage 1 (Data Preparation): ❶-❸ AutoCE takes as input a set of parameters (such as the number of tables/columns, domain size, skewness, and correlation), then generates a number of datasets. ❹-❻After that, each dataset is labeled by a unified cardinality estimation testbed, and each label is a score vector that measures the performance of the CE models.

Stage 2 (Training Phase I): ❶-❷ AutoCE extracts the CE-related features for each dataset through a process of feature engineering, and models the extracted features as a feature graph. To learn a graph encoder that can produce similarity-aware dataset embeddings w.r.t. the performance of the CE models, ❸-❽ AutoCE conducts the deep metric learning by recursively learning from batches of labeled feature graphs with loss back-propagation.

Stage 3 (Training Phase II): ❾-❿ AutoCE validates the performance of the trained graph encoder with all the training data. If certain feature graphs are not well-predicted, ⓫ AutoCE augments them with well-predicted feature graphs by combining their features and labels. Then, ⓬ AutoCE conducts incremental learning with the new training data to improve the generality of the advisor.

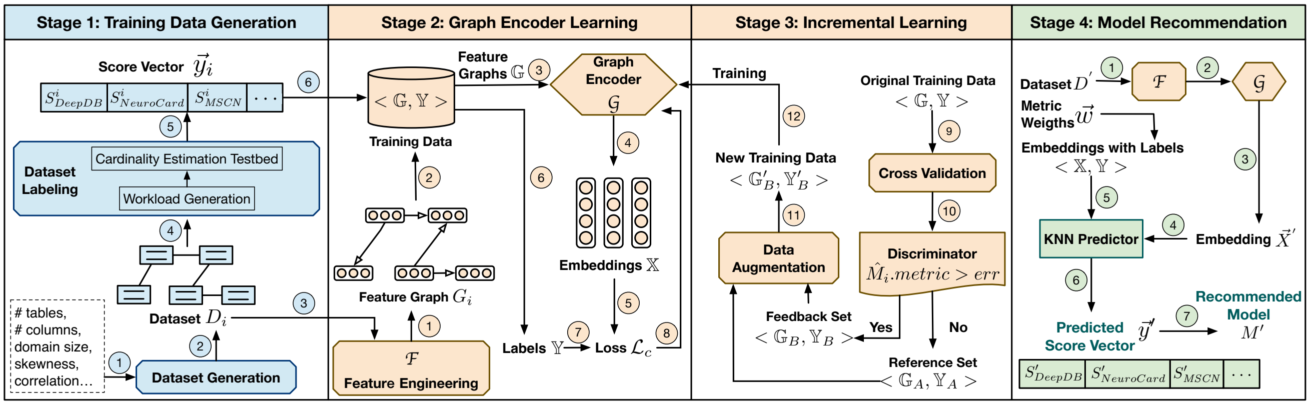


Fig. 3. An overview of AutoCE on model recommendation including data preparation (Stage 1), training (Stage 2-3) and recommendation (Stage 4).

Stage 4 (Recommendation): AutoCE makes the recommendation for a target dataset and specified weights of metrics. Specifically, ❶-❸ it goes through the process of feature engineering, acquires the embedding of the target dataset with the updated encoder, ❹-❺ and then searches for the k-nearest neighbors based on its distances to the embeddings of labeled datasets. Finally, ❻-❼ an averaged score vector is computed based on the neighbors' labels associated with the metric weights, and the top ranker of the score vector corresponds to the selected model.

A. *Training Data Generation*

(1) Dataset Generation. As existing datasets are hard to cover diverse features for CE, we present the *dataset generation* component, which can provide thousands of datasets to cover a relatively comprehensive space of data features. Particularly, it takes a set of input parameters, and then generates a set of synthetic tables with diverse distributions of data features. Theoretically speaking, it is rather hard to generate all meaningful data distributions. Nevertheless, AutoCE enables a good generalization regarding different datasets as it generates the data based on selected data features for cardinality estimation. To handle the unexpected data distributions, we design an online adaptive method which can detect the unexpected data distribution, then uses online learning to obtain the ground truth, and finally updates the model accordingly.

(2) Dataset Labeling. This component is an offline procedure that leverages a unified testbed to obtain the performance (e.g., Q-error and inference latency) of the CE models on each dataset efficiently. We normalize these performance values, and then combine them into a score vector (see Subsection IV-B2). For each combination of weighted performance metrics, each dataset D_i corresponds to a score vector $\vec{y}_i$, which has a length of m that indicates the performance of CE models $\mathbb{M}$. Note that the framework of AutoCE is extensible, so any newly-emerged CE model and specified metrics can be readily incorporated into the framework.

B. *Graph Encoder Learning*

(1) Feature Engineering. This component extracts the relevant data features, then represents them as feature graphs. Specifically, it extracts various features of columns of a single table, such as the skewness and cross-column correlations. Meanwhile, it also considers the join correlation across tables. Since a relational schema can be naturally represented as a graph where the nodes correspond to tables and edges correspond to join relations, we represent the extracted features of a dataset as a feature graph. In a feature graph, each node contains the extracted features of a single table, and each edge denotes a join between two tables. Consequently, the feature graphs $\mathbb{G}$ become the input of the graph encoder, and they will be transformed to the dataset embeddings $\mathbb{X}$.

(2) Learning the graph encoder using Deep Metric Learning (DML). AutoCE aims to learn a graph encoder to produce a dataset embedding that can capture the relation between diverse data features and the performance of different CE models. As discussed in Section I, it is a challenging problem as it is hard to learn the mapping from the datasets to the CE model's performance due to the large datasets-to-performance space. To tackle this problem, we define the similarity between datasets regarding the CE model's performance ($\mathbb{Y}$), then we leverage the high-level idea of deep metric learning that captures performance similarities/dissimilarities between the datasets. To be specific, given a dataset, it pushes together its embeddings x and its similar counterparts ($\mathbb{X}^+$), and pushes apart the embeddings x and its dissimilar counterparts ($\mathbb{X}^-$). The objective to learn an encoder such that $d(f(x), f(x^+)) >> d(f(x), f(x^-))$, where (x, x^+) is a positive pair and (x, x^-) is a negative pair, and $d(\cdot)$ can be the Euclidean distance. Since the number of dataset pairs is much larger than the number of datasets, our DML-based learning approach can well capture the relation from the datasets to the performance of CE models.

C. *Incremental Learning*

The incremental learning is an integral part of AutoCE and is used for co-training the recommendation model. The main purpose of incremental learning is to reduce the labeling overhead by augmenting the training data, thereby improving the robustness of the recommendation model. If the discriminator detects a poorly-predicated sample, it augments them with a well-predicted sample by combining their features and labels. Afterward, the graph encoder is incrementally trained with the new training data. Particularly, it conducts a cross-validation phase with the original training data $< \mathbb{G},\mathbb{Y} >$. For a training sample $< G_i,\vec{y_i} >$, if AutoCE recommends a CE model that produces a large error, (i.e., $\hat{M_i}.metric > err$), the discriminator assigns it to the feedback set $< \mathbb{G}_B$, $\mathbb{Y}_B >$. Otherwise, the sample is put to the reference set $< \mathbb{G}_A$, $\mathbb{Y}_A >$. For data augmentation, AutoCE generates a new feature graph G'_i with a synthetic label $\vec{y}'_i$ for each sample $G_i \in \mathbb{G}_B$. Specifically, it first searches for a nearest neighbor $G_j \in \mathbb{G}_A$ based on the Euclidean distance, then generates new training data $< G'_i,\vec{y}'_i >$ by combining the pair of feature graphs $< G_i, G_j >$ and pair of labels $< \vec{y}_i,\vec{y}_j >$. Consequently, AutoCE uses the new training data $< \mathbb{G}'_B,\mathbb{Y}'_B >$ to learn an updated graph encoder incrementally.

D. Model Recommendation

To select a model for a given dataset and specified metric weights $\vec{w}$, we develop a KNN-based predictor. Specifically, for a given new dataset D', AutoCE represents the datasets with feature extractor $\mathcal{F}$, utilizes the trained graph encoder $\mathcal{G}$ to output a similarity-aware embedding $\vec{X}'$, and feeds the embedding to the KNN predictor. The predictor searches for the k-nearest embeddings $\{\vec{X}_1, ., \vec{X}_k\}$ for the embedding $\vec{X}'$ based on Euclidean distance, then averages their labels $\{\vec{y}_1, ., \vec{y}_k\}$ to a unified score vector $\vec{y}'$. Finally, the model with the highest score in vector $\vec{y}'$ is the selected model M'.

IV. Dataset Generation and Labeling

A. Dataset Generation

The data generation method is divided into two parts: single-table generation and multi-table generation. We generate the datasets based on three data features (i.e., skewness, column correlation, and join correlation) as follows:

(F1) Skewness. The data for each column is generated using the Pareto distribution, which generates a set of random numbers with skewness. We change the skewness parameter $skew$ from 0 to 1, where $skew = 0$ means uniform distribution. As the $skew$ increases, the data becomes a more skewed distribution. Specifically, the probability density function (PDF) of the column skewness is defined as follows:

$$f(x) = \frac{(1 + x \cdot (skew - 1))^{-1-\frac{-1}{skew-1}}}{v_{max} - v_{min}} \tag{1}$$

where $skew$ is the skewness of a column, v_{max} is the maximum value, and v_{min} is the minimum value.

(F2) Column Correlation. We change the correlation parameter r to control the probability that two columns have the same value in the same position. That is, take two values (v_1, v_2) at the same position in the two columns, and make them equal with the probability of r. The larger the probability is, the stronger the correlation is. When the corr is 0, there is no correlation between two columns.

(F3) Join Correlation. We generate the join correlation for each PK-FK join by varying the probability p, Particularly, we generate a join correlation p within the range [j_{min}, j_{max}], then take a portion of p without replacement from the PK column of the *main table*. After that, we populate the FK column by randomly sampling values from the portion of data. As a result, the higher value of p, the larger portion of PK column data is occupied in the FK column data.

1) Single Table Generation: The generation of a single table takes input as these variables: the number of columns n, the number of rows k, domain size d, skewness parameter $skew$, max correlation r, then outputs a generated table with n columns. It works in two steps. First, it sets the max value v_{max} to d and sets v_{min} to 1, selects a $skew$ value within [0,1], then generates the column with k rows based on Equation 1. Second, it iterates over all the generated columns to add a correlation between the two selected columns. Particularly, for every two adjacent columns, we correct their correlation r to a random value between 0 and 1 by modifying their values.

2) Multi-Table Generation: Generating multiple tables is based on the single-table generation. AutoCE takes as input the table size n, the max and min join correlation j_{max} and j_{min}, then generates a correlated dataset with n tables in three steps. Firstly, it generates n tables independently using the single-table generation. Secondly, it selects m tables as *main tables*, and assigns a primary key to each main table. Thirdly, it randomly correlates an arbitrary table (could be a main table) with a main table by generating a PK-FK join with a correlation p following the procedure of F3.

B. Dataset Labeling

1) Model Training and Testing: We develop a CE testbed to measure the performance of CE models on each generated dataset. We implemented seven state-of-the-art CE models, including three query-driven methods, three data-driven methods, and one hybrid approach.The labeling process consists of four steps. First, it generates a query workload against the dataset. Second, it acquires the true cardinalities by running the queries in the database. Third, it trains the candidate CE models. For data-driven models, they are trained with the dataset directly to learn a joint distribution. For query-driven models, they are trained with a set of encoded training queries with true cardinalities. Finally, it measures their performance score with a set of testing queries and true cardinalities.

To incorporate a new cardinality estimation baseline into AutoCE, we deploy the baseline to the cardinality estimation testbed, which conducts the dataset labeling and produces the corresponding score vectors. Finally, AutoCE is able to select the suitable CE model based on the new score vectors.

2) Score Normalization: We use Q-$error$ [22] and inference latency T to measure the models' performance on accuracy and efficiency. We use the mean of Q-$error$, denoted as Q-$error_{mean}$ to represent the error of a certain CE model on the testing queries against a dataset. Note that it is possible to use other percentiles of the metrics, such as 50-th, 95-th, and 99-th of Q-$error$. In this work, we choose the mean as the metric. We also denote T_{mean} as the average estimation time of a model on the testing queries. We normalize the scores with different weights of Q-$error_{mean}$ and T_{mean} to consider various combinations of them. We have evaluated the

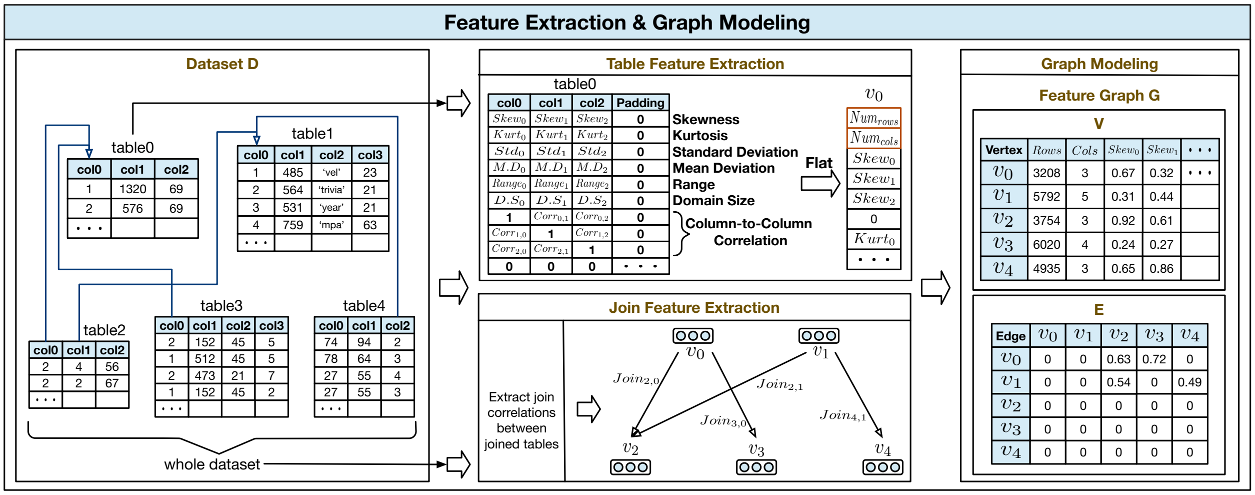


Fig. 4. Feature engineering for a dataset D, including the processes of feature extraction and graph modeling.

improvement of the combinations of $Q\text{-}error$ and T on the E2E latency (See Section VII-D).

Let S^{D_i,M_j} denotes the performance score of the j-th CE model M_j on the i-th dataset D_i, which is a combination of the accuracy score and the efficiency score. Let w_a (resp. w_e) be an accuracy weight (resp. efficiency weight), where $w_a + w_e = 1$ and $0 \leq w_a \leq 1$ with a step of 0.1. The score S^{D_i,M_j} is defined as follows:

$$S^{D_i,M_j} = w_a * S_a^{D_i,M_j} + w_e * S_e^{D_i,M_j} \tag{2}$$

where $S_a^{D_i,M_j}$ and $S_e^{D_i,M_j}$ denote its normalized accuracy score and efficiency score, respectively. Specifically, their corresponding formulas are follows:

$$S_a^{D_i,M_j} = \frac{Max(Q\text{-}error_{mean}^{D_i}) - Q\text{-}error_{mean}^{D_i,M_j}}{Max(Q\text{-}error_{mean}^{D_i}) - Min(Q\text{-}error_{mean}^{D_i})} \tag{3}$$

$$S_e^{D_i,M_j} = \frac{Max(T_{mean}^{D_i}) - T_{mean}^{D_i,M_j}}{Max(T_{mean}^{D_i}) - Min(T_{mean}^{D_i})} \tag{4}$$

In the above two formulas, $Max(Q\text{-}error_{mean}^{D_i})$ and $Max(T_{mean}^{D_i})$ represent the maximum $Q\text{-}error_{mean}$ and T_{mean} of all cardinality estimation models $\mathbb{M}$ against dataset D_i; $Q\text{-}error_{mean}^{D_i,M_j}$ and $T_{mean}^{D_i,M_j}$ represents the $Q\text{-}error_{mean}$ and T_{mean} of j-th cardinality estimation method M_j against the $i\text{-}th$ dataset.

Definition 1: (D-error). We define a metric, called $D\text{-}error$, based on the performance store $S^{D_i,M}$. The rationale of $D\text{-}error$ is to measure how far the performance score of M is from that of the optimal model M_{opt} regarding dataset D_i. The optimal model M_{opt} refers to the model with the highest score on the dataset. The formula is denoted as: $D\text{-}error = \frac{S^{D_i,M_{opt}} - S^{D_i,M}}{S^{D_i,M}}$, where $S^{D_i,M}$ is the performance score of model M on dataset D_i.

V. Model Training and Inference

A. Feature Engineering

1) Feature Extraction: Conventionally, the machine learning techniques encode the tuples, train a model, and do the inference for a tuple. In our task, a training sample is a dataset instead of a tuple, and `AutoCE` does the inference for a dataset each time. Therefore, we only need to capture the data distributions that are relevant to the overall performance of the cardinality estimation models. We extract three relevant features (i.e., column correlation, skewness, and domain size) for each column. We also use other related data features as complements, including kurtosis, range, and standard/mean deviation, as well as the number of rows and columns. We also consider the features of joins for multi-table datasets. Instead of using the join selectivity that often has a small value, we compute the join correlation by taking the set of the FK column data of a table, then calculating its ratio over the PK column data of a joined table. By doing so, the network model can better capture the join correlation. The extraction of skewness, column correlation, and join correlation is a reverse process of the data generation (See three processes (F1), (F2), and (F3) in Section IV-A).

2) Graph Modeling: After the features of a dataset have been extracted, we model them as a graph, called *feature graph*, in which the vertices of the graph correspond to the features of the table, and the edges of the graph correspond to joins across the tables; the weight on an edge is computed based on the join correlation between two tables. A feature graph consists of a *vertex matrix* and an *edge matrix*, and graph modeling involves vertex modeling and edge modeling.

(1) Vertex Modeling. For a dataset, we set the maximum number of columns of all tables to m. For each table, there are two unique features: the number of rows and the number of columns. For each column of a table, we extract k data features. For every two columns of a table, we extract a total of $m \times m$ correlation features. In total, a table can have a maximum of $(k + m) \times m + 2$ features. If a generated table has less than m columns, we use padding to fill the empty positions with 0. Consequently, a n-table dataset has a vertex matrix V with the shape of V is $[n, (k + m) \times m + 2]$.

(2) Edge Modeling. Suppose a dataset has n tables, an edge matrix E has the size of $n \times n$. If there is an FK in the $j\text{-}th$ table referencing the PK of the $i\text{-}th$ table, then the value of $E[i][j]$ stores the join correlation, otherwise $E[i][j] = 0$.

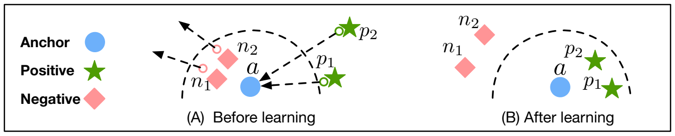
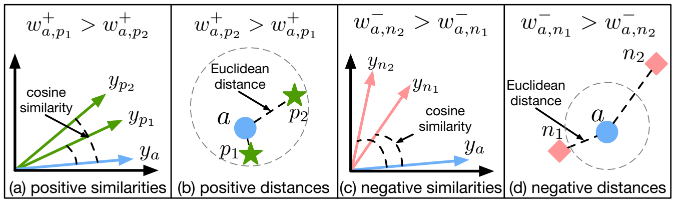


Fig. 5. Learning effect of graph contrastive learning.



(a) positive similarities (b) positive distances (c) negative similarities (d) negative distances

Fig. 6. Pair weighting: $w^+_{a,p1}$ and $w^-_{a,n1}$ denotes the weight for a positive pair $< a, p1 >$ and negative pair $< a, n1 >$, respectively.

Example 3: As shown in Figure 4, given a five-table dataset D, we extract features from each table with a maximum of 4 columns. The vertex features of table i are flattened as a vector v_i, i.e., v_0 for $table_0$. The vertex matrix V has the shape of $[5, (6 + 4) \times 4 + 2] = [5, 42]$, and the edge matrix E has a shape of 5×5 and contains four non-empty join weights. For instance, the FK column of table v_2 has a portion of 54% of the PK column of table v_1.

B. Encoding the Feature Graph

We encode the feature graphs using Graph Isomorphism Network (GIN) [36], which is a state-of-the-art graph neural network. GIN conducts nonlinear mapping of the feature graph through L `GINConV` layers. For `GINConV` layer, GIN multiplies the features of each vertex v' neighbors and the corresponding edge features between v and its neighbors, and then each vertex v aggregates the multiplication result, producing a representation of the vertex. Finally, GIN takes as input the representations of all vertices and outputs a representation using sum pooling.

Specifically, our graph encoder consists of L `GINConV` layers and one layer of sum pooling. It takes a *feature graph* $G = (V, E)$ as input and outputs an encoded vector $\vec{X}$. We denote H^l as the output of the l-th `GINConV` layer, where H^0 is the input of the first `GINConV` layer with ($H^0 = V$). Let h^l_i denote the feature vector of the i-th vertex output by the l-th `GINConV` layer, which is computed by:

$$h_i^{(l+1)} = f_\theta \left((1 + \epsilon) h_i^l + \sum\nolimits_{j \in N(i)} e'_{ji} \cdot h_j^l \right) \tag{5}$$

In the above formula, f_θ represents a function determined by a learnable parameter θ in the network; ϵ is also a learnable parameter; $N(i)$ represents the joined neighbors of vertex i, and e'_{ji} represents the join correlation between vertex j and i.

C. Learning the Graph Encoder

We learn the graph encoder by deep metric learning. Particularly, we treat the score vectors as labels, and use the labels to determine whether two feature graphs are in the same classes or not (See Eq. 6 and Eq. 7). Intuitively, the closer two score vectors are, the more likely two feature graphs are in the same class. Since we have the score vectors with various combinations (weights of w_a and w_e), our graph encoder can learn from each combination of score vectors to support various users' requirements (See Eq. 2).

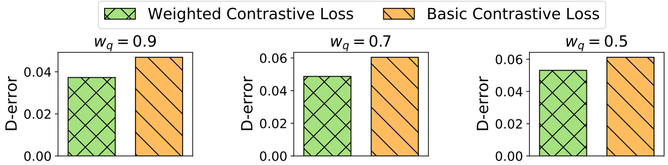


Fig. 7. Comparisons between two contrastive loss functions.

Example 4: Figure 5 illustrates the learning effect of deep metric learning. There are three types of graph instances: anchor, positive, and negative instances. Each instance is an embedding representation for a feature graph. After the learning process, the positive instances $\{p_1, p_2\}$ have been pulled closer to the anchor while the negative instances $\{n_1, n_2\}$ have been pushed away from the anchor a.

Measuring the similarity with score vectors. We use score vectors to decide whether two instances belong to the same class or not. We first define the similarity and label as follows:

Definition 2: (Performance Similarity). Given the labels $\vec{y}_i$ and $\vec{y}_j$ of *feature graphs* G_i and G_j, We use cosine similarity to define their performance similarity Sim_{ij} as follows:

$$Sim_{ij} = \frac{\vec{y}_i \cdot \vec{y}_j}{||\vec{y}_i|| \; ||\vec{y}_j||} \tag{6}$$

Definition 3: (Positive/Negative Instance). Given a pair of feature graphs $< G_i, G_i >$ and a threshold τ, we denote the G_i as an anchor, then j belongs to either the positive index set $\mathcal{P}_i$ or the negative index set $\mathcal{N}_i$ based on their similarity S_{ij} and threshold τ. Particularly, G_j is a positive instance of G_i if S_{ij} is greater than τ. Otherwise, G_j is a negative instance of G_i. The formula is defined as follows:

$$G_j.index \in \begin{cases} \mathcal{P}_i & Sim_{ij} \geq \tau \\ \mathcal{N}_i & Sim_{ij} < \tau \end{cases} \tag{7}$$

We define $U_{i,j}$ as the distance between the embeddings of the two *feature graphs* G_i and G_j. We use the Euclidean distance function to define the $U_{i,j}$ as follows:

$$U_{ij} = ||\vec{X}_i - \vec{X}_j||_2 \tag{8}$$

where $\vec{X}_i$ and $\vec{X}_j$ are the embeddings of feature graph G_i and feature graph G_j, respectively.

Weighted contrastive loss. We define a new loss function, called weighted contrastive loss, to take into account both the distance weight and similarity weight. To the best of our knowledge, no existing works [9], [27], [31] have considered the weight for label similarity. Intuitively, since we wish to capture the differences between graph pairs based on their similarities to an anchor, we can (1) assign larger weights to the positive pairs with larger similarities/distances, and (2) allocate larger weights to the negative pairs with smaller similarities/distances.

Example 5: Figure 6 illustrates four cases of pair weighting for either performance similarities or embedding distances. For case (a) where two positive instances p_1 and p_2 have different similarities, the larger the similarity ($Sim_{a,p1} > Sim_{a,p2}$), the more important the larger pair ($w^{+}_{a,p1} > w^{+}_{a,p2}$). For case (b) where two positive instances p_1 and p_2 have different distances, the larger the distance ($U_{a,p2} > U_{a,p1}$), the more important the larger pair ($w^{+}_{a,p2} > w^{+}_{a,p1}$). On the contrary, the negative pairs are more important if they have smaller similarities/distances since we would like to push them apart. For case (c), the smaller the similarity, the larger the weight ($w^{-}_{a,n2} > w^{-}_{a,n1}$). For case (d), the smaller the distance, the larger the weight ($w^{-}_{a,n1} > w^{-}_{a,n2}$).

Apart from the consideration of absolute similarity weight and distance weight, we also consider the relative weight. The intuition is similar to the Example 5 but using the relative weight: for a positive (resp. negative) pair $< G_i, G_j >$, the larger (resp. smaller) the relative similarity/distance, the more (resp. less) important the pair $< P_i, P_j >$ is. For instance, for a negative pair $< G_i, G_j >$ with a set of neighbors K, the smaller the relative similarity ($Sim_{ij} - Sim_{ik}$), the more important the pair $< G_i, G_j >$ is. By putting it all together, we define the weighted contrastive loss as follows:

Definition 4: (Weighted Contrastive Loss). Given a batch of m *feature graphs*, the loss $\mathcal{L}_c$ is defined as follows:

$$\mathcal{L}_c = \frac{1}{m}\sum_{i=1}^{m}\left(log \sum_{k\in\mathcal{P}_i}\left(e^{U_{ik}+Sim_{ik}}\right) + log \sum_{k\in\mathcal{N}_i}\left(e^{(\gamma-U_{ik}-Sim_{ik})}\right)\right) \tag{9}$$

where $\mathcal{P}_i$ and $\mathcal{N}_i$ are the positive index set and the negative index set of the anchor G_i; γ is a fixed margin.

To verify the effectiveness of the loss function in (9), we compare our weighted contrastive loss function with the basic contrastive loss [5] as follows:

$$\mathcal{L} = \frac{1}{m}\sum_{i=1}^{m}\left(\sum_{k\in\mathcal{P}_i} U_{ik} - \sum_{k\in\mathcal{N}_i} U_{ik}\right) \tag{10}$$

As shown in Figure 7, the new loss function produces a better result than that of the basic contrastive loss on 200 synthetic datasets.. The main reason is that our loss function considers both distance weight and similarity weight while the basic contrastive loss neglects such information.

Analysis of pair weighting. The optimization goal of the loss function is to shorten the embedding distance of the positive pairs, and enlarge the embedding distance of the negative pairs. By differentiating L_c on U_{ij}, then we compute the weights for the positive and negative pairs as follows:

$$w^{+}_{i,j} = \left|\frac{\partial L_c}{\partial U_{ij}}\right| = \frac{1}{\sum_{k\in\mathcal{P}_i} e^{[(U_{ik}-U_{ij})+(Sim_{ik}-Sim_{ij})]}} \tag{11}$$

$$w^{-}_{i,j} = \left|\frac{\partial L_c}{\partial U_{ij}}\right| = \frac{1}{\sum_{k\in\mathcal{N}_i} e^{[(U_{ij}-U_{ik})+(Sim_{ij}-Sim_{ik})]}} \tag{12}$$

where $w^{+}_{i,j}$ and $w^{-}_{i,j}$ denotes the weight for a positive pair and a negative pair, respectively. It is not difficult to observe that, as the relative distance ($U_{ij} - U_{ik}$) and the relative similarity ($Sim_{ij} - Sim_{ik}$) increase, the positive pairs have larger weights, and the negative pairs have smaller weights.

Description of the training algorithm. Algorithm 1 illustrates the procedures of the similarity-aware graph encoder learning.

Algorithm 1: DML-based Graph Encoder Learning

```
Input: All $n$ feature graphs with labels, batch_size $m$.
Output: A trained GIN $\mathcal{G}$.
1  Initialize GIN network parameters $\theta$ ;
2  for each epoch in training process do
3      for each batch in this epoch do
4          for $i \leftarrow 1$ to $m$ do
5              for $j \leftarrow 1$ to $m$ do
6                  Calculate $Sim_{i,j}$ according to Eq. 6 ;
7                  if $Sim_{i,j} \geq \tau$ then
8                      $\mathcal{P}_i$.add(j) ; // Positive index
9                  else
10                     $\mathcal{N}_i$.add(j) ; // Negative index
11             $\vec{X_i} = \mathcal{G}(G_i)$ // Embedding with Eq.5
12         Calculate $\mathcal{L}_c$ according to Eq. 9 ;
13         $\theta = \theta - \eta \cdot \nabla\mathcal{L}_c$ ; // learning rate $\eta$
14 return $\mathcal{G}$
```

We initialize the GIN network parameters θ (Line 1). For each epoch (Line 2), a set of n feature graphs are divided into $\lceil \frac{n}{m} \rceil$ batches. The training steps for each batch are as follows. First, it computes the similarity $Sim_{i,j}$ of each graph pair $< G_i,G_j >$ (Line 3-6). For each anchor G_i, if $Sim_{i,j}$ is greater than a threshold τ, the index of the instance G_j is assigned to the positive index set $\mathcal{P}_i$ (Line 7-8). Otherwise, it is added to the negative index set $\mathcal{N}_i$ (Line 9-10). Second, for each instance G_i, the embedding $\vec{X_i}$ is generated based on Eq. 5 (Line 11). Then, it calculates the weighted contrastive loss with the entire batch according to Eq. 9 (Line 12). Third, it updates the parameters θ of GIN $\mathcal{G}$ via the backpropagation of the loss $\mathcal{L}_c$ (Line 13). When all epochs have been conducted, it returns the trained GIN $\mathcal{G}$ (Line 14).

D. Recommendation using Learned Encoder

A KNN-based predictor. With a trained GIN $\mathcal{G}$, we can acquire an embedding for an arbitrary dataset. The key idea is to compare the distances between the labeled embeddings to the unlabeled target dataset's embedding, then find the k-nearest neighbors' labels associated with the users' requirements. We define the recommendation candidate set as follows:

Definition 5: (Recommendation Candidate Set (RCS)). Given a trained GIN network $\mathcal{G}$ and n feature graphs $\mathbb{G}$ with labels $\mathbb{Y}$, the recommendation candidate set is $(\mathbb{X}, \mathbb{Y})$ which contains all the embeddings of $\mathbb{G}$ with labels $\mathbb{Y}$.

Recommendation process. AutoCE makes the recommendation in three steps. First, it feeds all n training feature graphs into the trained GIN $\mathcal{G}$ to generate their embeddings $\mathbb{X}$. Since these feature graphs have been labeled in the data preparation phase, their labels $\mathbb{Y}$ are also available. Hence, the *RCS* $(\mathbb{X}, \mathbb{Y})$ is obtained. Second, AutoCE takes a target set D' as input, produces its feature graph G', and generates its embedding $\vec{X}'$ with the trained GIN $\mathcal{G}$. After that, it searches for its k-nearest neighbors $\{\vec{X}_1, \ldots, \vec{X}_k\}$ in the *RCS* based on their embeddings $\mathbb{X}$'s distances to the target embedding $\vec{X}'$. Then the k-nearest neighbors' indexes are added to the set $\mathbb{K}$. Third, AutoCE averages the labels associated with $\mathbb{K}$ into a score

Algorithm 2: Incremental Learning with Mixup

```
Input: Original training data $< \mathbb{G}, \mathbb{Y} >$, threshold $b$, GIN $\mathcal{G}$.
Output: An updated GIN $\mathcal{G}$.
1  $\mathbb{X} = \mathcal{G}(\mathbb{G})$ ; // Get all Embeddings
2  Split $< \mathbb{G}, \mathbb{Y}, \mathbb{X} >$ to $< \mathbb{G}_1, \mathbb{Y}_1, \mathbb{X}_1 >, ..., < \mathbb{G}_\xi, \mathbb{Y}_\xi, \mathbb{X}_\xi >$ ;
3  for $v$ in $[1, \xi]$ do
4      Get a validation set $(\mathbb{X}_v, \mathbb{Y}_v)$ ;
5      $\mathbb{X}_c, \mathbb{Y}_c = \mathbb{X} \setminus \mathbb{X}_v, \mathbb{Y} \setminus \mathbb{Y}_c$ ; // Get RCS
6      for $X_i, G_i$ in $\mathbb{X}_v, \mathbb{G}_v$ do
7          $\mathbb{K} = KNN(\vec{X_i}, \mathbb{X}_c, k)$ ;
8          $\hat{M_i} = Recommend(\mathbb{Y}_c, \mathbb{K})$ ; // Eq.13
9          if $\hat{M_i}.D\text{-}error > b$ then
10             $\mathbb{G}_B.add(G_i), \mathbb{Y}_B.add(\vec{y_i})$ ;
11         else
12             $\mathbb{G}_A.add(G_i), \mathbb{Y}_A.add(\vec{y_i})$ ;
13 for $G_i, \vec{y_i}$ in $\mathbb{G}_B, \mathbb{Y}_B$ do
14     Get the nearest neighbor $G_j$ and $\vec{y_j}$ in $\mathbb{G}_A, \mathbb{Y}_A$ ;
15     $G'_i, \vec{y}'_i = Mixup(G_i, \vec{y_i}, G_j, \vec{y_j})$ ; // Eq.14
16     $\mathbb{G}'_B.add(G_i), \mathbb{Y}'_B.add(\vec{y_i})$ ; // Get new sample
17 Incrementally train a $\mathcal{G}$ according to Algorithm 1 ;
18 return $\mathcal{G}$ ;
```

vector, and outputs the top ranker's corresponding model as the recommended model. The formula is given as follows:

$$M' = \max(\frac{\sum_{i=1}^{k} \vec{y}_{\mathbb{K}[i]}}{k}).index \tag{13}$$

where $\mathbb{K}$ is the indexes of the k-nearest neighbors, and $\vec{y}$ is a vector of the corresponding labels; M' is the recommended model that has a max score in the score vector.

E. Online Adapting for Unexpected Data Distribution

To handle the unexpected data distribution, we design an online adaptive method: it detects the data distributions that are out of the trained distributions. Then, it uses the online learning to select the CE model, and finally it updates the model accordingly. More specifically, the online adaptive method works in three steps. First, given a target dataset, it detects if there is any data drift, i.e., a data distribution is disparate from the existing RCS based on the feature graph's Euclidean distance. We set a distance threshold to determine if a dataset is apart from the RCS. Particularly, the distance from a dataset D to the RCS is defined as the closest distance from D to all the feature graphs in RCS. We sort the distance values of the feature graphs in RCS in ascending order, then take the 90th percentile distance value as the threshold. If the distance of a dataset is greater than the threshold, it is considered as an unexpected data distribution. Second, for an unexpected dataset, it employs the online learning to get the ground truth, then adds the new labeled sample to the RCS. Third, it uses the new training sample to update the recommendation model by deep metric learning.

VI. Incremental Learning

A. Incremental Learning with Data Augmentation

Incremental learning process. The incremental learning process has three steps: (1) collect the feedback and reference; (2) augment the data; and (3) train a new encoder. We introduce the procedures in Algorithm 2 with the following steps:

Step1. `AutoCE` takes as input the training data $< \mathbb{G}, \mathbb{Y} >$, a threshold b, and a trained GIN $\mathcal{G}$. First, it generates all embeddings $\mathbb{X}$ of $\mathbb{G}$ with GIN $\mathcal{G}$ (Line 1). Then it equally splits the training data with embeddings to ξ folds. Second, it adopts a cross-validation approach to collect the feedback and references (Line 3-12). Specifically, it recursively takes a validation set $(\mathbb{X}_v, \mathbb{Y}_v)$, and reserves the rest of the folds as the recommendation candidate set (RCS). For each instance $X_v \in \mathbb{X}_v$, it conducts a validation process to collect the feedback and reference by the KNN-based prediction (Line 7), it produces a recommended model $\hat{M_i}$ (Line 8). If the model $\hat{M_i}$ has a D-error greater than the threshold b, then its feature graph and label are added to the feedback set $< \mathbb{G}_B, \mathbb{Y}_B >$ (Line 9-10). Otherwise, it goes to the reference set $< \mathbb{G}_A, \mathbb{Y}_A >$ (Line 11-12).

Step2. For each sample $< G_i, \vec{y_i} >$ in the feedback set $< \mathbb{G}_B, \mathbb{Y}_B >$, `AutoCE` uses *Mixup* to generate a new sample (Line 13-16). Specifically, it finds the nearest neighbor $< G_j, \vec{y_j} >$ in the reference set based on the Euclidean distance. Then it generates a new sample $< G'_i, \vec{y}'_i >$ by employing *Mixup* to incorporate linear interpolations into their feature vectors and labels, respectively. The formula is as follows:

$$\begin{aligned} G'_i &= \lambda G_i + (1 - \lambda) G_j \\ \vec{y}'_j &= \lambda \vec{y_i} + (1 - \lambda) \vec{y_j} \end{aligned} \tag{14}$$

where $\lambda \sim Beta(\alpha, \beta)$, a Beta distribution within [0, 1].

Step3. With the new training samples $< \mathbb{G}_B, \mathbb{Y}_B >$ and the original training data, `AutoCE` incrementally trains a more robust GIN $\mathcal{G}$ (Line 17-18).

VII. Experiments

A. Experiment Setting

Datasets. As shown in Table I, we use two real-world datasets and 1,200 synthetic datasets.

(1) Real-world datasets. We evaluate `AutoCE` on two real-world datasets: a movie-rating dataset IMDB [29] and a STATS [7] dataset from the Stack Exchange network. We name them IMDB-light and STATS-light (See Table I). We use them as unseen testing datasets to verify the effectiveness of `AutoCE`. Since each dataset offers only one testing sample for `AutoCE`, we adopt a split approach to obtain more testing samples. Specifically, we randomly select 20 testing samples from each dataset in two steps: (1) randomly select 1-5 joined tables from the dataset with the join keys; (2) randomly select 1-2 non-key columns for each chosen table. We name them as `IMDB-20` and `STATS-20`.

(2) Synthetic datasets. The generation of synthetic datasets is described in Section IV-A. We generate 1,200 synthetic datasets where 1000 datasets are used as the training set, and 200 datasets are kept unseen for testing `AutoCE`.

Training and Inference Time. With all the training data (1000 labeled datasets), `AutoCE` takes 107s to train the graph encoder in an offline manner. For the testing set, the average inference time of `AutoCE` is 0.79s for each dataset.

TABLE I
STATISTICS OF DATASETS.

Dataset	#Table	#Row	#Column	Total domain size
IMDB-light	6	2.1K-339K	12	4.3×10^4
STATS-light	8	1K-328K	23	1.9×10^5
Synthetic	1-5	10K-50K	2-25	1.6×10^4

Workloads. We generate 10,000 SPJ queries similar to [37], [38], from which 9,000 queries are used for training each query-driven method and 1,000 queries are kept for testing. We also use the CEB-IMDB benchmark [21], where we use all the query templates, but for the ease of implementation, we eliminated the 'GROUP BY' and 'LIKE' predicates in queries.

Metrics. We use *D-error* in Definition 1 to measure the recommendation error regarding a dataset. Particularly, we use a small *D-error* threshold (e.g., 0.1) to indicate if a CE model is accurately recommended or not. We also consider the metrics of Q-error, inference latency, recommendation efficiency, and end-to-end latency.

Environment. All experiments were performed on a server with a 20-core Intel(R) Xeon(R) 6242R 3.10GHz CPU, an Nvidia Geforce 3090ti GPU, and 256GB DDR4 RAM.

Model Selection Baselines. We compare with 4 baselines.

(1) MLP-based Selection. We implement an MLP-based baseline by appending a multilayer perceptron of three layers to the GIN network. This baseline treats the model selection problem as a classification task and trains the recommendation model with the cross-entropy loss [26].

(2) Rule-based Selection. We design a rule-based selection: (i) randomly selecting data-driven models for single-table datasets, and (ii) randomly choosing query-driven models for multi-table datasets.

(3) Knn-based Selection. We develop a Knn-based baseline that extracts the features of datasets, then directly uses the KNN predictor based on the distances of the features rather than the embeddings to select models.

(4) Sampling-based Selection. We implement an online learning baseline. Specifically, it samples a portion of a dataset, trains and tests all the CE models against the samples, and finally selects the best-performing CE model.

Cardinality Estimation Baselines. We compare AutoCE with 9 baselines for cardinality estimation as follows:

(1) MSCN [11]. This method encodes the query features with a table set, a join set, and a predicate set using one-hot vectors. Then, it utilizes the multi-set convolutional network to learn a mapping function from the feature set to the cardinality.

(2) LW-XGB [4]. It employs a tree-based ensemble method, called XGBoost [1], which encodes a query as a sequence of selection ranges, then learns a mapping function from the query encoding to the predicted cardinality.

(3) LW-NN [4]. This approach is based on a lightweight fully connected neural network, and its query encoding method and training strategy are the same as LW-XGB.

(4) DeepDB [8]. This method relies on relational sum-product networks. It divides a table into row clusters and column clusters. Then it utilizes sum nodes (resp. product nodes) to combine the row clusters (resp. column clusters).

(5) BayesCard [35]. This method relies on Bayes Network to learn a joint distribution.

(6) NeuroCard [37]. It builds a deep autoregressive model on the samples of the full-outer joins of the base tables, then conducts a progressive sampling to make the estimates.

(7) UAE [33]. This method proposes differentiable progressive sampling via the Gumbel-Softmax trick to enable learning from queries. Then it unifies both query and data information using the deep autoregression model [38].

(8) An ensemble CE method that takes the weighted average estimation of all the CE models (the weight is proportional to their performance on the training datasets).

(9) A default PostgreSQL CE estimator.

B. Evaluation of Recommendation Efficacy

Comparing AutoCE with 4 selection baselines. As shown in Figure 8, we compare the efficacy of AutoCE with MLP and Rule on the synthetic datasets by varying accuracy weights from 1.0 to 0.1. Figure 8(c) shows an overall evaluation of the model performance with D-error, and Figure 8(a) and Figure 8(b) illustrate the breakdown of Q-error and latency, respectively (e.g., the D-error subgraph of $w_a = 0.9$ in (c) is broken down to the Q-error subgraph of $w_a = 0.9$ in (a) and the Latency subgraph of $w_e = 0.1$ in (b)). The results clearly show that AutoCE outperforms MLP, Rule, Sampling and Knn by using deep metric learning and incremental learning. It can be seen that (1) when the accuracy weight is between 0.5 and 1.0, the D-error distribution of AutoCE is significantly smaller than that of MLP, Rule, Sampling and Knn. The mean of D-error of AutoCE is 2.5x, 6.7x, 2.8x, and 4.9x better than MLP, Rule, Sampling and Knn respectively. (2) the mean of Q-error of AutoCE is 1.2x, 1.7, 1.3x, and 1.6x better than MLP, Rule, Sampling, and Knn respectively. (3) when the accuracy weight is between 0 and 0.3, the mean latency of AutoCE is 1.6x better than MLP. (4) Rule has the worst performance with different weights, indicating that the general selection is insufficient to select the CE models for various datasets and performance metrics. (5) Sampling is inferior to AutoCE as the performance of CE models has a high variance regarding different samples of datasets. (6) Knn is worse than AutoCE because it simply relies on the distances of data features rather than the distances of embeddings that can be aware of the CE models' performance.

For the real-world datasets, the Q-error of AutoCE significantly outperforms the other four baselines. As shown in Figure 10, we compare the D-error of AutoCE with the other four baselines. On IMDB-20, the mean of D-error of AutoCE is 3.2x, 12.7x, 2.9x, and 9.7x better than MLP, Rule, Sampling, and Knn respectively. On STATS-20, the mean of D-error of AutoCE is 2.4x, 7.1x, 1.6x, and 4.5x better than MLP, Rule, Sampling and Knn respectively.

Regarding CEB-IMDB benchmark, the data-driven cardinality estimators are rather difficult to implement due to the involved large number of tables. For example, 24G GPU memory is not enough for training UAE, and it takes 50 hours to train a NeuroCard model for one dataset. Hence, we

TABLE II
RECOMMENDATION ACCURACY OF AutoCE AND TWO BASELINES OVER 200 SYNTHETIC DATASETS AND 40 REAL-WORLD DATASETS.

		(a) $w_a = 1.0$			(b) $w_a = 0.9$			(c) $w_a = 0.7$		
Datasets	Advisor	Accuracy (ϵ = 0.1)	Accuracy (ϵ = 0.15)	Accuracy (ϵ = 0.2)	Accuracy (ϵ = 0.1)	Accuracy (ϵ = 0.15)	Accuracy (ϵ = 0.2)	Accuracy (ϵ = 0.1)	Accuracy (ϵ = 0.15)	Accuracy (ϵ = 0.2)
Synthetic (200)	MLP-based	56%	66%	74%	55%	69.5%	78%	45%	60%	72%
	Rule-based	16%	23%	31.5%	20%	28%	37%	28%	37.5%	43%
	Knn-based	44%	50%	61%	34%	45.5%	58.5%	37.5%	48.5%	59.5%
	Sampling	44%	51%	53.5%	22.5%	28.5%	37%	18.5%	31.5%	41%
	AutoCE	**81.5%**	**86%**	**87.5%**	**89%**	**94%**	**97%**	**76.5%**	**88%**	**96%**
IMDB-20	MLP-based	55%	70%	75%	50%	75%	80%	55%	65%	70%
	Rule-based	10%	20%	20%	20%	20%	20%	30%	40%	40%
	Knn-based	40%	40%	50%	45%	55%	55%	50%	60%	80%
	Sampling	25%	25%	30%	15%	20%	25%	15%	15%	25%
	AutoCE	**80%**	**85%**	**95%**	**90%**	**95%**	**100%**	**80%**	**90%**	**100%**
STATS-20	MLP-based	45%	50%	60%	40%	60%	65%	50%	65%	75%
	Rule-based	5%	10%	15%	10%	10%	20%	15%	35%	35%
	Knn-based	45%	65%	70%	50%	55%	65%	50%	70%	75%
	Sampling	45%	55%	55%	30%	40%	45%	10%	20%	20%
	AutoCE	**85%**	**85%**	**90%**	**70%**	**80%**	**85%**	**85%**	**90%**	**90%**

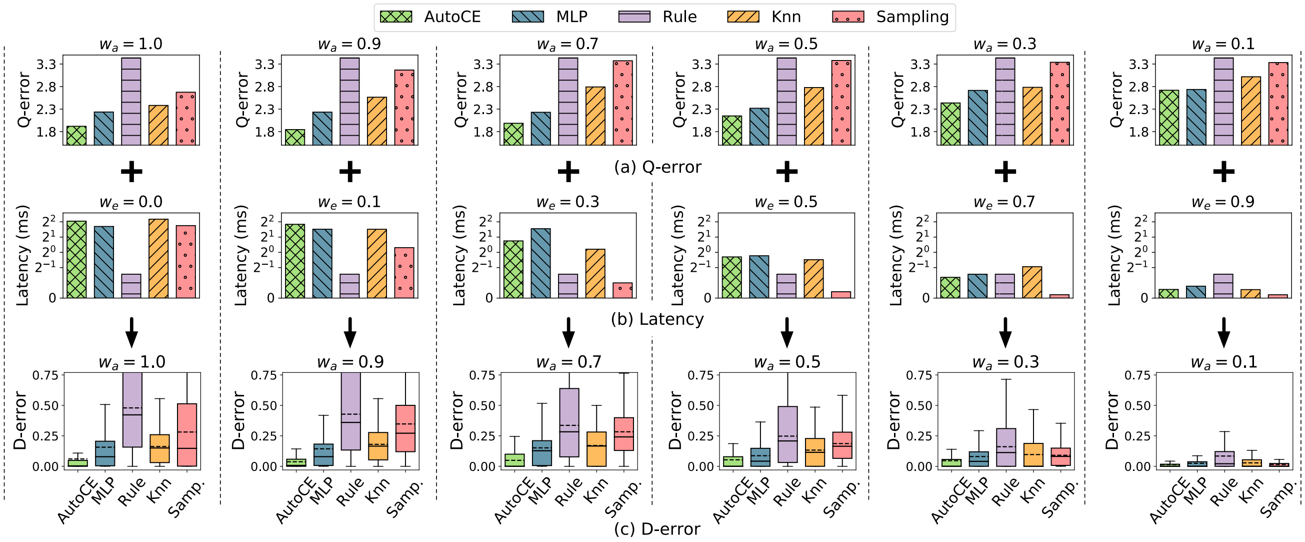


Fig. 8. Comparisons between AutoCE and four selection strategies.

TABLE III
EVALUATION OF EFFICACY ON CEB BENCHMARK.

Method	AutoCE	MSCN	LW-NN	LW-XGB
D-error$_{w_a=1.0}$	**0%**	1.30%	7.86%	100%
D-error$_{w_a=0.9}$	**3.16%**	11.80%	7.08%	90.76%
D-error$_{w_a=0.7}$	**2.51%**	26.29%	5.502%	82.81%
D-error$_{w_a=0.5}$	**1.74%**	40.78%	3.93%	74.87%

TABLE IV
AUTOCE'S D-ERROR UNDER DIFFERENT k.

k	1	2	3	4	5
D-error$_{w_a=1.0}$	7.67%	**6.04%**	6.74%	7.01%	8.71%
D-error$_{w_a=0.9}$	5.43%	**3.73%**	3.94%	4.80%	5.89%
D-error$_{w_a=0.7}$	6.51%	**4.87%**	5.20%	5.24%	6.17%
D-error$_{w_a=0.5}$	6.62%	**5.31%**	5.89%	6.37%	6.94%

just conduct the experiments with the query-driven cardinality estimators including MSCN, LW-NN, and LW-XGB. As shown in Table III, AutoCE always achieves the lowest recommendation error with various accuracy weight w_a, demonstrating the effectiveness of AutoCE. The error of MSCN decreases as w_a decreases, while LW-NN does the opposite. This is mainly because MSCN is more accurate on most query templates than LW-NN, but the inference efficiency is not as good as LW-NN. We found LW-XGB has a high D-error because it has both a low accuracy and a low inference efficiency.

Comparing AutoCE with 9 CE baselines. As shown in Figure 9, we compare AutoCE with 9 CE baselines. It is clearly visible that the D-error distribution of AutoCE is lower than the fixed CE models for different accuracy weight w_a. Particularly, the mean of D-error of AutoCE is 5.2% while the average mean of D-error of other fixed CE models is 38.2%. In addition, AutoCE has 2.8x and 12.3x better model performance than the best-performed one (i.e., DeepDB) and the worst-performed one (i.e., LW-XGB), respectively. These results verify that a fixed CE model or the ensemble method cannot perform well on different types of datasets. What's more, they cannot maintain a lower D-error when the accuracy weight changes. For example, since UAE favors high estimation accuracy by learning from data and queries, it has a higher D-error when w_a decreases. On the contrary, as LW-NN has a higher estimation efficiency because of the lightweight neural networks, it has a lower D-error as w_e increases. Although the ensemble strategy is relatively stable by combining the CE methods, it has a large margin of D-error to AutoCE (20%).

Comparing AutoCE with varying k. We compare the recom-

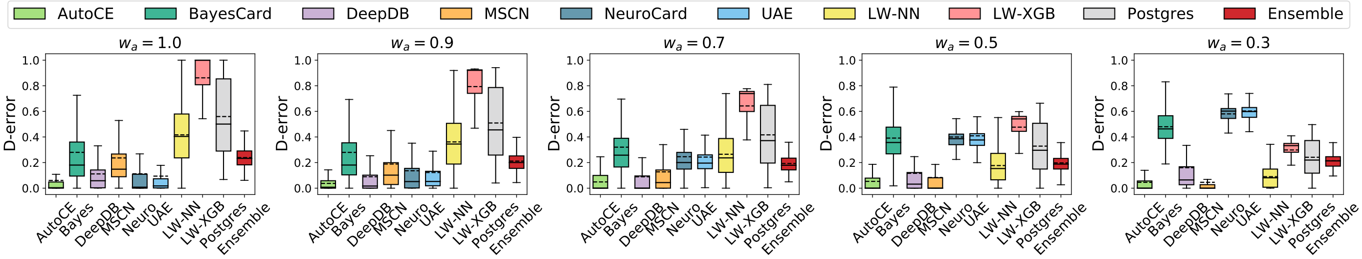
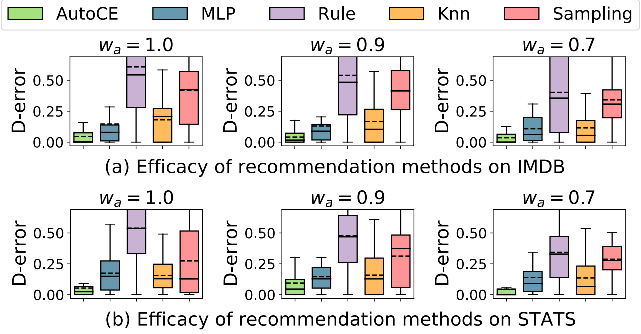


Fig. 9. Comparisons between AutoCE and seven CE models (plus a PostgreSQL estimator and an ensemble method) on the synthetic datasets.



(a) Efficacy of recommendation methods on IMDB

(b) Efficacy of recommendation methods on STATS

Fig. 10. Evaluation of efficacy on real-world datasets.

mendation quality (i.e., D-error) with different k. Particularly, we vary k from 1 to 5, and evaluate the recommendation error on synthetic datasets with different accuracy weight w_a from 0.5 to 1.0. As shown in Table IV, D-error is always the smallest when $k = 2$. When $k = 1$, the performance of the KNN predictor is largely affected by the nearest embedding, which may cause a greater error. When $k \geq 3$, the performance of the predictor is worse because there could be embeddings that are far away from the embedding of the target dataset.

C. Evaluation of Recommendation Accuracy

As shown in Table II, we evaluate the recommendation accuracy of five approaches on the synthetic and real-world datasets. We vary three small threshold values (0.1, 0.15, 0.2) of D-error and three values (1.0, 0.9, 0.7) of the accuracy weight w_a. Overall, it clearly shows that AutoCE outperforms the four baselines in all settings. On average, the accuracy of AutoCE is 1.4x, 2.8x, 1.8x, 2.4x higher than that of MLP, Rule, Sampling and Knn respectively on the synthetic dataset. For the real-world dataset, the accuracy of AutoCE is 1.4x, 4.2x, 1.5x, 3.1x higher than that of MLP, Rule, Sampling and Knn. We have two observations. First, AutoCE can make the recommendation of high accuracy (on average 85%) on the real-world datasets, which proves that AutoCE works on the real-world datasets by using the feature-driven learning method. Second, AutoCE outperforms MLP with 55% higher accuracy, indicating the effectiveness of its core parts, i.e., deep metric learning and incremental learning.

D. Impact on Query Optimization

We evaluate the end-to-end latency in PostgreSQL v13.1 with different CE models. Particularly, we use the modified PostgreSQL code of [7] to inject the cardinalities into the query optimizer. To be specific, we invoke each CE model to estimate the cardinalities of all sub-plan queries for a query, then feed them into the query optimizer to generate the query plan. Finally, we execute the query plan in PostgreSQL to get the end-to-end latency. We use 30 synthetic datasets including 15 single-table datasets and 15 multi-table (2-5 tables) datasets, and we run 100 queries against each dataset.

TABLE V
END-TO-END LATENCY IN POSTGRESQL.

	Single-table	Multi-table	Single.	Multi.
Method	Running time + Inference Latency		Improvement	
PostgreSQL	22.11s + 1.78s	1.732h + 16s	-	-
TrueCard	**20.99s**	**1.168h**	**12.14%**	**32.74%**
BayesCard	21.68s + 6.78s	1.461h + 60s	-19.13%	14.90%
DeepDB	21.65s + 5.03s	1.345h + 36s	-11.68%	22.45%
MSCN	21.72s + 0.33s	1.344h + 3s	7.70%	22.55%
NeuroCard	21.58s + 13.73s	1.265h + 125s	-47.80%	25.15%
UAE	21.66s + 13.07s	1.283h + 122s	-45.37%	24.16%
LW-NN	21.78s + 0.01s	1.778h + 0.1s	8.80%	-2.40%
LW-XGB	21.77s + 0.40s	1.828h + 4s	7.20%	-5.34%
AutoCE w_a=0.5	**21.38s + 0.16s**	1.319h + 18s	**9.84%**	23.75%
AutoCE w_a=1.0	21.29s + 2.99s	**1.247h + 41s**	-1.63%	**27.56%**

Table V shows the total running time of workloads with the model inference latency. For single-table datasets, we find that (1) the estimated cardinalities mainly affect the scan operators of the query plans, e.g., sequential scans or index scans, and (2) the inference latency could be a major factor for the overall execution time. For instance, the inference latency takes more than half of the overall time for NeuroCard and UAE while the query-driven model LW-NN takes only 10ms. AutoCE ($w_a = 0.5$) achieves the largest improvement (∼9.84%) as it takes into account both accuracy and efficiency. For multi-table datasets, the plan quality is the major factor for query optimization where the inference latency takes only a small fraction of the overall time. Particularly, more accurate estimated cardinalities result in a better join order and suitable join operators, e.g., hash joins or nested-loop joins. AutoCE ($w_a = 1$) outperforms other estimators in performance (∼27.56%), because AutoCE can select the most accurate models for different kinds of datasets.

E. Ablation of Deep Metric Learning

We conduct an ablation study on DML. We implement AutoCE (Without DML) by appending three fully connected layers to the GIN network and use the MSE loss function $L = \sum_{i=1}^{m} ||\vec{y}_i - \vec{y}||_2$ to train the the entire network. Then it can recommend a CE model using $max(\vec{y}).index$.

As shown in Figure 11, we compare the D-error of AutoCE and AutoCE (Without DML) with accuracy weights (0.9, 0.7, 0.5). The result on the synthetic datasets indicates that (1) the D-error of AutoCE is always lower than that of AutoCE (Without DML) as the accuracy weight w_a decreases. (2) the

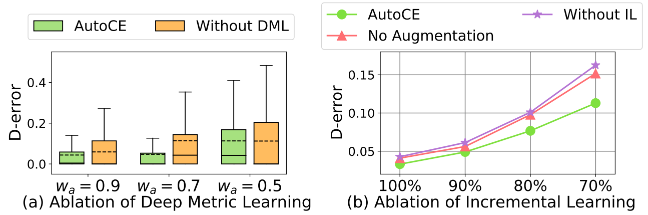


Fig. 11. Ablation on core parts of AutoCE.

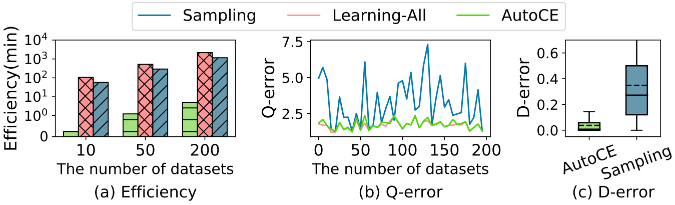


Fig. 12. Comparisons between AutoCE and online learning methods.

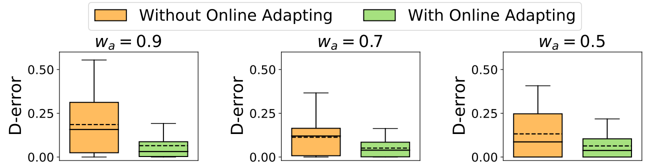


Fig. 13. Ablation of Online Adapting.

mean of D-error of AutoCE is 40% better than that of AutoCE (Without DML), which verifies the effectiveness of DML.

F. Ablation of Incremental Learning

Effectiveness. In Figure 11, we compare the performance of AutoCE with AutoCE (Without IL) and AutoCE (No Augmentation) using an accuracy weight $w_a = 0.9$. The x-axis is the percentage of used training data, and the y-axis is the mean of D-error. We can see that AutoCE always has a lower D-error with respect to different proportions of the training data. The main reason is that it can obtain more information through incremental learning. When using 70% amount of training data, AutoCE has ∼5% and ∼4% smaller D-error than that of AutoCE (Without IL) and AutoCE (No Augmentation).

Efficiency. Mixup-based incremental learning can synthesize new training samples without labeling. Given 93% training data, we generate (∼15%) new training samples with a D-error threshold of 10% on the feedback, then incrementally train a new encoder with the synthetic data. The results show that it has the same D-error as that of AutoCE (Without IL) exploiting 100% training samples, indicating that AutoCE saves about 7% of the dataset labeling time (∼2 hours).

G. Comparing AutoCE with Online Learning Methods

We compare AutoCE with two online learning methods over 200 datasets. Particularly, the sampling method selects a model by training and testing all the candidate models against a sample of each dataset, and the learning-all (LA) method uses the full dataset each time. As shown in Figure 12a, two online learning methods incur a significant overhead as the number of datasets increase (LA and sampling approach takes 2230 and 1200 minutes for 200 disparate datasets, respectively), while AutoCE requires less than 5 minutes to complete such a task. Figure 12b depicts that the Q-error of AutoCE is close to the LA method (both have a Q-error of 1.8, but the sampling method has a fluctuated Q-error distribution regarding disparate datasets. Figure 12c shows that the D-error of AutoCE is 3.7% w.r.t. the LA method while the sampling method has a D-error of 34.8%. In summary, AutoCE not only achieves near-optimal CE performance but also reduces the selection overhead significantly (by 455x).

H. Ablation of Online Adapting

To verify the effectiveness of the online adaptive approach, we conducted an experiment: (1) we randomly generate datasets and encode them as feature graphs. (2) we deliberately select 100 datasets as the unexpected data distribution based on the distances. (3) we use the 100 datasets for online adapting. We compare the performance of AutoCE with an ablation study on the online adaptive method. As shown in Figure 13, it can be found that our approach effectively reduces recommendation error by more than 1x on average for the datasets with the unexpected data distributions.

VIII. Related Work

CE Model Selection. Several empirical studies [7], [10], [23], [29], [32] on learned cardinality estimation have been conducted. They analyzed the performance of many ML-based CE models, followed by general rules and insights for selecting the models. For instance, [29] suggests query models are more effective for multiple tables, indicating that users could select a query-driven model for multi-table datasets. [7] concludes that only data-driven methods such as FLAT [53], DeepDB, and BayesCard, can improve the end-to-end performance of the PostgreSQL baseline, meaning that users could select a data-driven model in production. Unfortunately, such general insights are insufficient to select a CE model for diverse datasets. As shown in the experiments, either query-driven or data-driven models could outplay other CE models, depending on the datasets and metrics.

Deep Learning Models for Database. There are currently many works applying deep learning models in databases [12], [18], [25], [34], [39], [41]–[43], [49], [50], [52]. For example, using Monte Carlo tree search to rewrite queries [51], using graph neural network to generate materialized views [6], and the hybrid learning-based and cost-based query optimizer [40].

IX. Conclusion

We propose a learned model advisor, AutoCE. We leverage deep metric learning to train a graph encoder, which can capture the relation from diverse features of datasets to the performance of cardinality estimation (CE) models. Moreover, we employ the graph encoder and a distance-aware predictor to recommend a learned CE model for a target dataset. We also propose an approach of feedback-driven data augmentation to generate new training samples without labeling the datasets. Experimental results have shown that AutoCE significantly outperformed the baselines on both accuracy and efficiency. Moreover, AutoCE achieved the best performance compared with other CE models in PostgreSQL.

References

[1] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *SIGKDD*, pages 785–794, 2016.

[2] B. Dageville, T. Cruanes, and et al. The snowflake elastic data warehouse. In *SIGMOD*, pages 215–226. ACM, 2016.

[3] P. Das, N. Ivkin, T. Bansal, L. Rouesnel, P. Gautier, Z. S. Karnin, L. Dirac, L. Ramakrishnan, A. Perunicic, I. Shcherbatyi, W. Wu, A. Zolic, H. Shen, A. Ahmed, F. Winkelmolen, M. Miladinovic, C. Archambeau, A. Tang, B. Dutt, P. Grao, and K. Venkateswar. Amazon sagemaker autopilot: a white box automl solution at scale. In *DEEM@SIGMOD*, pages 2:1–2:7. ACM, 2020.

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

[5] R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality reduction by learning an invariant mapping. In *CVPR*, volume 2, pages 1735–1742. IEEE, 2006.

[6] Y. Han, C. Chai, J. Liu, G. Li, C. Wei, and C. Zhan. Dynamic materialized view management using graph neural network.

[7] Y. Han, Z. Wu, P. Wu, R. Zhu, J. Yang, L. W. Tan, K. Zeng, G. Cong, Y. Qin, A. Pfadler, Z. Qian, J. Zhou, J. Li, and B. Cui. Cardinality estimation in DBMS: A comprehensive benchmark evaluation. *CoRR*, abs/2109.05877, 2021.

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

[9] E. Hoffer and N. Ailon. Deep metric learning using triplet network. In *SIMBAD*, volume 9370, pages 84–92. Springer, 2015.

[10] K. Kim, J. Jung, I. Seo, W. Han, K. Choi, and J. Chong. Learned cardinality estimation: An in-depth study. In *SIGMOD*, pages 1214–1227. ACM, 2022.

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

[12] H. Lan, Z. Bao, and Y. Peng. A survey on advancing the DBMS query optimizer: Cardinality estimation, cost model, and plan enumeration. *Data Sci. Eng.*, 6(1):86–101, 2021.

[13] G. Li, H. Dong, and C. Zhang. Cloud databases: New techniques, challenges, and opportunities. *Proc. VLDB Endow.*, 15(12):3758–3761, 2022.

[14] G. Li and C. Zhang. HTAP databases: What is new and what is next. In *SIGMOD*, pages 2483–2488. ACM, 2022.

[15] G. Li and X. Zhou. Machine learning for data management: A system view. In *ICDE*, pages 3198–3201. IEEE, 2022.

[16] G. Li, X. Zhou, and L. Cao. AI Meets Database: AI4DB and DB4AI. In *SIGMOD*, pages 2859–2866, 2021.

[17] G. Li, X. Zhou, and L. Cao. Machine learning for databases. *VLDB*, 14(12):3190–3193, 2021.

[18] G. Li, X. Zhou, J. Sun, X. Yu, Y. Han, L. Jin, W. Li, T. Wang, and S. Li. opengauss: An autonomous database system. *Proc. VLDB Endow.*, 14(12):3028–3041, 2021.

[19] Z. Liu, C. Chen, X. Yang, J. Zhou, X. Li, and L. Song. Heterogeneous graph neural networks for malicious account detection. In *CIKM*, pages 2077–2085. ACM, 2018.

[20] Y. Lu, S. Kandula, A. C. König, and S. Chaudhuri. Pre-training summarization models of structured datasets for cardinality estimation. *Proc. VLDB Endow.*, 15(3):414–426, 2021.

[21] R. Marcus, A. Kipf, A. van Renen, M. Stoia, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Flow-loss: Learning cardinality estimates that matter. *Proceedings of the VLDB Endowment*, 14(11), 2021.

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

[23] P. Negi, R. C. Marcus, A. Kipf, H. Mao, N. Tatbul, T. Kraska, and M. Alizadeh. Flow-Loss: Learning Cardinality Estimates That Matter. *Proc. VLDB Endow.*, 14(11):2019–2032, 2021.

[24] D. Nigenda, Z. Karnin, M. B. Zafar, R. Ramesha, A. Tan, M. Donini, and K. Kenthapadi. Amazon sagemaker model monitor: A system for real-time insights into deployed machine learning models. In *KDD*, pages 3671–3681. ACM, 2022.

[25] Y. Peng, B. Choi, and J. Xu. Graph learning for combinatorial optimization: A survey of state-of-the-art. *Data Sci. Eng.*, 6(2):119–141, 2021.

[26] R. Rubinstein. The cross-entropy method for combinatorial and continuous optimization. *Methodology and computing in applied probability*, 1(2):127–190, 1999.

[27] H. O. Song, Y. Xiang, S. Jegelka, and S. Savarese. Deep Metric Learning via Lifted Structured Feature Embedding. In *CVPR*, pages 4004–4012. IEEE Computer Society, 2016.

[28] J. Sun and G. Li. An End-to-End Learning-based Cost Estimator. *Proc. VLDB Endow.*, 13(3):307–319, 2019.

[29] J. Sun, J. Zhang, Z. Sun, G. Li, and N. Tang. Learned cardinality estimation: A design space exploration and a comparative evaluation. *Proc. VLDB Endow.*, 15(1):85–97, 2021.

[30] J. Wang, C. Chai, J. Liu, and G. Li. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.*, 15(1):72–84, 2021.

[31] X. Wang, X. Han, W. Huang, D. Dong, and M. R. Scott. Multi-similarity loss with general pair weighting for deep metric learning. In *CVPR*, pages 5022–5030. Computer Vision Foundation / IEEE, 2019.

[32] X. Wang, C. Qu, W. Wu, J. Wang, and Q. Zhou. Are we ready for learned cardinality estimation? *Proc. VLDB Endow.*, 14(9):1640–1654, 2021.

[33] P. Wu and G. Cong. A unified deep model of learning from both data and queries for cardinality estimation. In *SIGMOD*, pages 2009–2022, 2021.

[34] S. Wu, Y. Li, H. Zhu, J. Zhao, and G. Chen. Dynamic index construction with deep reinforcement learning. *Data Sci. Eng.*, 7(2):87–101, 2022.

[35] Z. Wu, A. Shaikhha, R. Zhu, K. Zeng, Y. Han, and J. Zhou. Bayescard: Revitilizing bayesian frameworks for cardinality estimation. *arXiv preprint arXiv:2012.14743*, 2020.

[36] K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018.

[37] Z. Yang, A. Kamsetty, S. Luan, E. Liang, Y. Duan, P. Chen, and I. Stoica. Neurocard: One cardinality estimator for all tables. *Proc. VLDB Endow.*, 14(1):61–73, 2020.

[38] Z. Yang, E. Liang, A. Kamsetty, C. Wu, Y. Duan, X. Chen, P. Abbeel, J. M. Hellerstein, S. Krishnan, and I. Stoica. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.*, 13(3):279–292, 2019.

[39] X. Yu, C. Chai, G. Li, and J. Liu. Cost-based or learning-based? A hybrid query optimizer for query plan selection. *Proc. VLDB Endow.*, 15(13):3924–3936, 2022.

[40] X. Yu, C. Chai, G. Li, and J. Liu. Cost-based or learning-based? a hybrid query optimizer for query plan selection. *Proceedings of the VLDB Endowment*, 15(13):3924–3936, 2022.

[41] X. Yu, G. Li, C. Chai, and N. Tang. Reinforcement learning with tree-lstm for join order selection. In *ICDE*, pages 1297–1308. IEEE, 2020.

[42] H. Yuan and G. Li. A survey of traffic prediction: from spatio-temporal data to intelligent transportation. *Data Sci. Eng.*, 6(1):63–85, 2021.

[43] H. Yuan, G. Li, L. Feng, J. Sun, and Y. Han. Automatic view generation with deep learning and reinforcement learning. In *ICDE*, pages 1501–1512. IEEE, 2020.

[44] C. Zhang. Parameter curation and data generation for benchmarking multi-model queries. In *VLDB PhD Workshop*, volume 2175, 2018.

[45] C. Zhang and J. Lu. Selectivity estimation for relation-tree joins. In *SSDBM*, pages 8:1–8:12. ACM, 2020.

[46] C. Zhang and J. Lu. Holistic evaluation in multi-model databases benchmarking. *Distributed Parallel Databases*, 39(1):1–33, 2021.

[47] C. Zhang, J. Lu, P. Xu, and Y. Chen. Unibench: A benchmark for multi-model database management systems. In *TPCTC*, volume 11135, pages 7–23. Springer, 2018.

[48] L. Zhang, C. Chai, X. Zhou, and G. Li. Learnedsqlgen: Constraint-aware SQL generation using reinforcement learning. In *SIGMOD*, pages 945–958. ACM, 2022.

[49] L. Zhang, C. Chai, X. Zhou, and G. Li. Learnedsqlgen: Constraint-aware SQL generation using reinforcement learning. In *SIGMOD*, pages 945–958. ACM, 2022.

[50] X. Zhou, C. Chai, G. Li, and J. Sun. Database meets artificial intelligence: A survey. *IEEE Trans. Knowl. Data Eng.*, 34(3):1096–1116, 2022.

[51] X. Zhou, G. Li, C. Chai, and J. Feng. A learned query rewrite system using monte carlo tree search. *Proceedings of the VLDB Endowment*, 15(1):46–58, 2021.

[52] X. Zhou, J. Sun, G. Li, and J. Feng. Query performance prediction for concurrent queries using graph embedding. *Proc. VLDB Endow.*, 13(9):1416–1428, 2020.

[53] R. Zhu, Z. Wu, Y. Han, K. Zeng, A. Pfadler, Z. Qian, J. Zhou, and B. Cui. FLAT: Fast, Lightweight and Accurate Method for Cardinality Estimation. *Proc. VLDB Endow.*, 14(9):1489–1502, 2021.