PACE: Poisoning Attacks on Learned Cardinality Estimation

Jintao Zhang
Tsinghua University
China
zjt21@mails.tsinghua.edu.cn

Chao Zhang*
cycchao@tsinghua.edu.cn
Tsinghua University
China

Guoliang Li*
liguoliang@tsinghua.edu.cn
Tsinghua University
China

Chengliang Chai
ccl@bit.edu.cn
Beijing Institute of Technology
China

ABSTRACT

Cardinality estimation (CE) plays a crucial role in database optimizer. We have witnessed the emergence of numerous learned CE models recently which can outperform traditional methods such as histograms and samplings. However, learned models also bring many security risks. For example, a query-driven learned CE model learns a query-to-cardinality mapping based on the historical workload. Such a learned model could be attacked by poisoning queries, which are crafted by malicious attackers and woven into the historical workload, leading to performance degradation of CE.

In this paper, we explore the potential security risks in learned CE and study a new problem of poisoning attacks on learned CE in a black-box setting. There are three challenges. First, the interior details of the CE model are hidden in the black-box setting, making it difficult to attack the model. Second, the attacked CE model's parameters will be updated with the poisoning queries, i.e., a variable varying with the optimization variable, so the problem cannot be modeled as a univariate optimization problem and thus is hard to solve by an efficient algorithm. Third, to make an imperceptible attack, it requires to generate poisoning queries that follow a similar distribution to historical workload. We propose a poisoning attack system, PACE, to address these challenges. To tackle the first challenge, we propose a method of speculating and training a surrogate model, which transforms the black-box attack into a near-white-box attack. To address the second challenge, we model the poisoning problem as a bivariate optimization problem, and design an effective and efficient algorithm to solve it. To overcome the third challenge, we propose an adversarial approach to train a poisoning query generator alongside an anomaly detector, ensuring that the poisoning queries follow similar distribution to historical workload. Experiments show that PACE reduces the accuracy of the learned CE models by 178×, leading to a 10× decrease in the end-to-end performance of the target database.

*Chao Zhang and Guoliang Li are the corresponding authors.

Authors' addresses: Jintao Zhang, Tsinghua University, China, zjt21@mails.tsinghua.edu.cn; Chao Zhang, cycchao@tsinghua.edu.cn, Tsinghua University, China; Guoliang Li, liguoliang@tsinghua.edu.cn, Tsinghua University, China; Chengliang Chai, ccl@bit.edu.cn, Beijing Institute of Technology, China.

This work is licensed under a Creative Commons Attribution International 4.0 License.

© 2024 Copyright held by the owner/author(s).

ACM 2836-6573/2024/2-ART37
https://doi.org/10.1145/3639292

CCS CONCEPTS

• **Information systems → Data management systems**.

KEYWORDS

Poisoning Attacks, Learned Models, Cardinality Estimation

ACM Reference Format:
Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2024. PACE: Poisoning Attacks on Learned Cardinality Estimation. *Proc. ACM Manag. Data* 2, 1 (SIGMOD), Article 37 (February 2024), 16 pages. https://doi.org/10.1145/3639292

1 INTRODUCTION

Learned cardinality estimation. Cardinality estimator is a vital component of the database query optimizer. In recent years, learned cardinality estimation (CE) methods [6, 9, 15, 17, 19, 44, 45, 48, 51, 52, 54] have attracted significant attention due to their higher performance than traditional estimation methods such as histograms and sampling. However, learning-based models incur the risks of being attacked as the training data could be poisoned to degrade the estimation performance. In this work, we take query-driven cardinality estimation models [6, 17, 19, 36], which are trained by fitting a set of training queries to their true cardinalities, as examples to study how to attack learned CE models by crafting poisoning queries. We discuss the attacks on data-driven CE models in Section 8.

Motivation. Nowadays, learned query-driven CE models have been deployed in real commercial systems [24, 25, 55, 56], such as Microsoft Scope [47], Amazon Redshift AutoWLM [38], GaussDB (for openGauss) [7, 28]. Normally, machine learning models in online systems likely update themselves for maintaining high accuracy when some new training data are arrived [14, 29, 47]. Similarly, query-driven CE models update themselves incrementally with newly executed queries [40, 46, 47]. This mechanism presents an opportunity for malicious people to craft some poisoning queries to attack the CE model, degrading the performance of the query optimizer. Unfortunately, existing studies primarily focus on improving the performance of CE models while neglecting their potential vulnerabilities to poisoning attacks. To the best of our knowledge, this is the first work that studies the poisoning attack on learned CE models.

Real scenarios of poisoning attacks in the context of databases. Let us consider two motivating examples, an "internal" case (Case 1) and an "external" case (Case 2). The attackers in both cases have the incentives to poison a DB model.

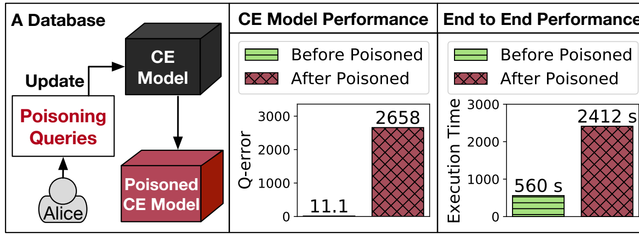

Figure 1: A example of a poisoning attack on a learned cardinality estimator.

Case 1 (Malicious Employee). *Suppose a scenario where an employee feels dissatisfied due to the unfair treatment or a notice of dismissal by his/her company. Since the company has a learning-based database used in production, s/he decides to perform a hidden act of retaliation. However, due to the company's strict permission policies, s/he has no deletion privilege and has only privilege of executing SELECT SQL queries for operation and maintenance. To this end, s/he decides to attack the company's database with poisoning queries.*

Case 2 (Malicious Competitor). *Consider a situation where a cloud vendor wants to beat its competitor in order to get better reputation. This cloud vendor deliberately rents a cloud database from its competitor and crafts malicious poisoning queries to attack the database's cardinality estimator in order to undermine the performance of the rented cloud database by poisoning attack. In this case, the cloud vendor not only has a strong incentive to carry out the attack but also has the authority to execute SQL queries on the target database.*

In addition, it's worth noting that previous research, such as that conducted by [20], has explored the issue of poisoning attacks on internal learned models in databases. Their study focused on poisoning attacks on learned indexes in a white box setting. In contrast, we study the poisoning attack in a black box setting (lack of all information about the target model), and there are more applications in realistic scenarios. In conclusion, we believe that the poisoning attacks on learned databases are a crucial topic that deserves more in-depth study.

Poisoning attack on learned CE. Considering a CE model used for estimating cardinalities of a given set of testing queries. The problem objective is to craft a small set of poisoning queries that, once used to update the CE model, would result in the model's lowest average estimation accuracy on the given test workload. As shown in Figure 1, suppose Alice intends to attack the cardinality estimator in the database. She can craft some poisoning queries and execute them, causing the cardinality estimator in the database to be updated with these queries. As a result, for a same set of test queries, the average Q-error of the estimator increased from 11.1 to 2658, and the end-to-end execution time of the database increased from 560 seconds to 2412 seconds.

Challenges. There are three challenges in poisoning attack on learned CE models in a black-box setting.

First, the black-box setting of learned models prevent us learning how to generate poisoning queries by the updating direction of the CE model's interior parameters.

Second, even under a white-box attack setting, efficiently solving the problem is difficult due to the updating of the CE model's parameters with the poisoning queries. In other words, a parameter of the optimization objective is changing with the optimization variable.

Third, there may be a significant divergence between the distribution of the poisoning queries and the historical workload, which could be easily detected [21]. Therefore, we need to generate queries that not only have poisoning effectiveness but also follow a similar distribution to the historical workload.

Our approach. To address these challenges, we propose a poisoning attack system PACE that can attack learned query-driven cardinality estimators in a black-box setting. To address the first challenge, we propose a method for speculating the model type of the black box by comparing the similarities of the black-box model and candidate models' performance, followed by training a surrogate model based on the speculated model type. This enables us to convert the black-box attack into a near-white-box attack. To address the second challenge, we propose to model the poisoning problem as a bivariate (i.e., poisoning queries and the CE model's parameters) optimization problem, and to achieve the optimization objective efficiently, we design an effective algorithm that utilizes a progressive update strategy to avoid unnecessary updates.

To address the third challenge, we train an anomaly detector that can identify anomaly queries. We then employ an adversarial approach to train a poisoning query generator alongside the detector, ensuring that the distribution of the poisoning queries is similar to that of the historical workload.

Contributions: We make the following contributions.

(1) We study a new problem of poisoning attacks on learned cardinality estimation models in a black-box setting.
(2) We propose a method of speculating and training a surrogate model to transform black-box attack into a near-white-box attack.
(3) We model the poisoning problem as a bivariate optimization problem, and design an algorithm that utilizes a progressive update strategy to achieve the optimization objective efficiently.
(4) We propose an adversarial approach to train a poisoning query generator alongside a trained anomaly detector, ensuring that poisoning queries follow a similar distribution to historical workload.
(5) We conducted extensive experiments, showing that our method reduces the accuracy of the learned CE models by 178×, leading to a 10× decrease in the end-to-end performance of the database. And PACE surpasses a basic algorithm by improving training efficiency by 9.7×, and enhances the normality of poisoning queries by 72%.

2 PRELIMINARIES

2.1 Query-driven Cardinality Estimation

We focus on poisoning attacks over query-driven cardinality estimators [6, 17, 19, 36]. Given a set of queries $\mathbb{Q} = \{q_0, q_1, \cdots, q_n\}$ with their cardinalities $\mathbb{Y} = \{y_0, y_1, \cdots, y_n\}$, query-driven cardinality estimators will represent each query as a vector x. Then the training data of a query-driven cardinality estimator will be a set of pairs (x, y). Finally, it learns a mapping from the query representations to the true cardinalities, which is regarded as a regression problem.

Table 1: Notations.

Notation	Description	Notation	Description
$f_{w_b}(\cdot)$	Black-box model	$\mathcal{L}$	Loss function of CE model
$f_s(\cdot)$	Surrogate model	$\mathcal{L}_s$	Loss function of $f_s(\cdot)$
$f_{w_p}(\cdot)$	Poisoned model	x	Encoding of a query
$\mathcal{D}$	Anomaly detector	$\mathcal{L}_d$	Loss function of $\mathcal{D}$
$\mathbb{D}_{\text{test}}$	Testing workload	$\mathbb{X}_p$	Poisoning queries
$\mathbb{Z}$	Gaussian noise	$\mathcal{G}$	Poisoning query generator

Formally, given a training workload $\mathbb{D}_{\text{train}} = \{\mathbb{Q}, \mathbb{Y}\}$, and a loss function $\mathcal{L}$, a query-driven CE model $f_w(x)$ is trained by an empirical risk minimization [43] strategy, and finally the optimal parameter w_b of CE model $f(\cdot)$ is obtained:

$$w_b \in \arg\min_{w} \sum_{(x,y)\in\mathbb{D}_{\text{train}}} \mathcal{L}(f_w(x), y) \tag{1}$$

where $\mathcal{L}$ is Q-error [33] loss, a most commonly used loss function in cardinality estimation. $\mathcal{L}(f_w(x), y) = \frac{max(f_w(x),y)}{min(f_w(x),y)}$, where $f_w(x)$ is the estimated cardinality of a query and y is the ground truth. $f_w(x)$ and y are both greater than 0, because the last activation layer of the CE model limits the normalized value of $f_w(x)$ in $(0, 1)$, and queries with $y = 0$ will be eliminated during the training phase. This problem is usually solved by gradient descent [37].

2.2 Threat Model

Adversary's goal: The attacker's objective is to craft *poisoning queries* $\mathbb{X}_p$ that can decrease the estimation accuracy of the target cardinality estimation model $f_w(\cdot)$ if it is updated using $\mathbb{X}_p$. The estimation accuracy refers to the Q-error [33] of $f_w(\cdot)$ on a given test set $\mathbb{D}_{\text{test}}$, $\sum_{(x,y)\in\mathbb{D}_{\text{test}}} Q\text{-}error(f_w(x), y)$.

Adversary's knowledge: We focus on *black-box* attacks where the attackers cannot acquire the model type and specific parameters w of the cardinality estimation model, and cannot get access to the data of the database and the training queries of the cardinality estimation model. The attackers can only obtain the database schema information to craft legal queries.

Adversary's capacity: Attackers are able to get the true labels $\mathbb{Y}$ (i.e., cardinalities) of crafted queries by executing COUNT(*) SQLs and can inject poisoning queries as the training queries of the cardinality estimation model. Moreover, attackers can obtain the estimated cardinalities $f_{w_b}(x)$ of the cardinality estimation model using the "*Explain*" command.

Attack evaluation metrics. We use four metrics as follows: **(1)** *Q-error* [33] is a metric for evaluating the accuracy of a cardinality estimation model. **(2)** *E2E latency* is used to quantify the end-to-end latency of query response in a database when utilizing a cardinality estimation model. **(3)** *Train_Time* is used to evaluate the training time of the poisoning queries generation algorithm. **(4)** *Divergence* is used to evaluate the normality of the poisoning queries distribution. Specifically, we use Jensen-Shannon Divergence [30] between the encodings of poisoning and historical queries. The higher the *Q-error* and *E2E latency* are, the more effective the attack is. The lower the *Train_Time* of the algorithm and the *Divergence* are, the more successful the attack is.

2.3 Problem Definition

Poisoning Query Generation Problem. The studied problem can be formally defined as follows: Given a trained black-box CE model $f_{w_b}(\cdot)$, a testing workload $\mathbb{D}_{\text{test}}$. The studied problem is to craft a small poisoning workload $\mathbb{X}_p$ that can decrease the CE model's estimation accuracy if $f_{w_b}(\cdot)$ is updated to $f_{w_p}(\cdot)$ using $\mathbb{X}_p$. The objective is to maximize the estimation error of $f_{w_p}(\cdot)$ over $\mathbb{D}_{\text{test}}$:

$$\mathbb{X}_p^* \in \arg\max_{\mathbb{X}_p} \left(\mathcal{F}(\mathbb{X}_p) = \sum_{(x,y)\in\mathbb{D}_{\text{test}}} \mathcal{L}(f_{w_p}(x), y) \right) \tag{2}$$

In this work, we leverage the gradient information of the CE models with respect to the poisoning queries to carry out our attacks. This methodology supports attacking all query-driven CE models that are based on neural networks.

2.4 Related Work

In the field of artificial intelligence security, many works [3, 5, 31, 34, 49, 50, 58] attack machine learning models by tampering with the features or labels of the training data. These works typically adopt a white-box setting, meaning that the type and parameters of the victim model are known. In the field of databases, [20] studied the problem of a poisoning attack on learned indices in a white-box setting. However, in a real-world system, the learned model is essentially a black box to potential attackers, meaning that both the training data and the learned model itself are entirely inaccessible. Furthermore, even under a white-box setting, attacking the learned CE model remains challenging. First, existing works are unsuitable to attack learned CE models. For instance, [20] only considers the linear regression model. And most works [3, 5, 31, 34, 49, 50, 58] can only produce poisoning samples of fixed dimension (e.g., fixed-size images). However, queries in learned CE models exhibit diverse join patterns that require specific treatment. Second, learned models in real-world scenarios are evolving, so it is necessary to craft poisoning queries efficiently. Otherwise, the attack could be obsolete. Third, to make an imperceptible attack, the poisoning queries should follow a similar distribution to the historical workload, otherwise it could be easily detected [21].

3 PACE FRAMEWORK

We first provide an overview of PACE in Section 3.1. Then, we describe the acquisition of surrogate CE model in Section 3.2 and poisoning query generation in Section 3.3. Finally, we introduce how to use PACE to attack the CE estimator in Section 3.4.

3.1 Overview

System Workflow. As shown in Figure 2, the workflow of the PACE can be summarized into three stages.

(a) Surrogate model acquisition. To cope with the black-box CE model, we propose to simulate the black-box model with a surrogate CE model $f_s(\cdot)$ by observing the input (queries) and output (estimated cardinalities) of the black-box model.

(b) Poisoning Data Generator.

Given the surrogate model $f_s(\cdot)$ which is a white-box model to us, we treat the attack on $f_s(\cdot)$ as an attack on the original black-box model $f_{w_b}(\cdot)$ so that the parameter w_b can be regarded as visible. To

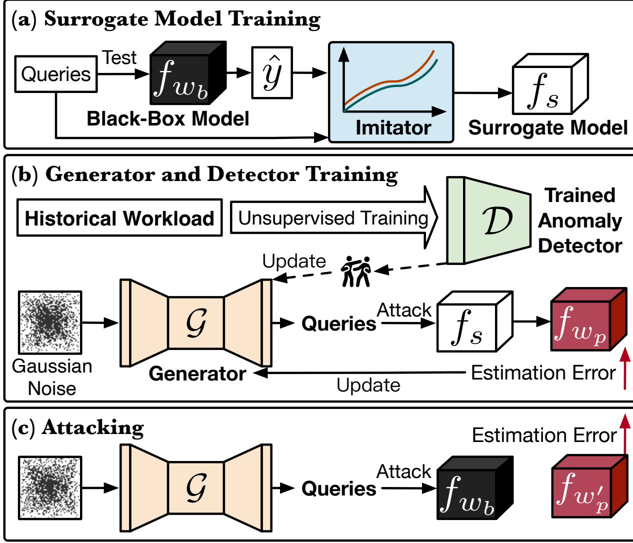

Figure 2: System overview. (§3)

achieve the optimization goal in Equation 2, we deploy a generator to generate poisoning queries.

The basic idea is to train the generator $\mathcal{G}$ with the objective of maximizing the estimation error of the poisoned surrogate model $f_{w_p}(\cdot)$. At the same time, to avoid generating queries that are rather different from the historical workload, the generator will also fight against an anomaly detector that can detect abnormal queries compared to historical queries.

(c) Attacking. The generated poisoning queries will be executed in the database, and the black-box CE model will be updated based on these queries, leading to larger estimation error.

3.2 Surrogate CE Model Acquisition

The key for simulating a model is its type (CNN, RNN, etc.) and parameters. To derive a surrogate model $f_s(\cdot)$ according to the black-box $f_{w_b}(\cdot)$, we first speculate the type of $f_{w_b}(\cdot)$, followed by acquiring the parameters via training with the output from $f_{w_b}(\cdot)$.

Specifically, we train six CE models [6, 17, 19, 36] that contain all types of query-driven models based on neural networks (See subsection 7.1). Next, we test these models as well as $f_{w_b}(\cdot)$ over some crafted queries. Finally, we select the model type that performs most similarly (including the accuracy and efficiency) to $f_{w_b}(\cdot)$.

For the parameters of $f_s(\cdot)$, given the model type, an intuitive solution is to take the query encoding x with the estimated result $f_{w_b}(x)$ of the black-box model as input, and trains a surrogate model according to a loss function like $\mathcal{L}(f_s(x), f_{w_b}(x))$ such that $f_s(\cdot)$ will be close to $f_{w_b}(\cdot)$. Unfortunately, solely relying on the output of the $f_{w_b}(\cdot)$ can lead to

poor generalization performance on unseen queries.

To overcome this problem, we incorporate the ground-truth labels into the training process of model $f_{w_b}(\cdot)$, which can help $f_s(\cdot)$ to imitate $f_{w_b}(\cdot)$ better because $f_{w_b}(\cdot)$ also contains the information of these labels. Therefore, we propose to use both outputs of $f_{w_b}(\cdot)$ and the ground-truth labels as training examples.

As a result, the trained surrogate model $f_s^*(\cdot)$ has a better generalization performance in imitating $f_{w_b}(\cdot)$.

The details are introduced in Section 4. The experimental results (See Section 7.4) indicate that, the parameters of the surrogate model are highly similar to that of the black box model after the simulation process, meaning that attacking $f_s^*(\cdot)$ is almost equivalent to attacking $f_{w_b}(\cdot)$.

Remark. Our method can be easily extended to support new CE models. When a new CE model needs to be considered, we only need to expand the k candidate models to $k+1$ candidate models.

3.3 Generator and Detector Training

We propose a generation-based approach to produce the poisoning queries. To generate diverse queries, we take random Gaussian noise as input for the generator. In particular, the poisoning workload $\mathbb{X}_p$ can be generated by feeding Gaussian noise set $\mathbb{Z} = \{\vec{z}|\vec{z} \sim \mathcal{N}(0,1)\}$ to the generator $\mathcal{G}$ as follows.

$$\mathbb{X}_p = \mathcal{G}(\mathbb{Z}) \tag{3}$$

Consequently, our goal becomes how to train the generator $\mathcal{G}$ to generate poisoning queries that can maximize the loss of the poisoned CE model $f_{w_p}(\cdot)$.

Therefore, the optimization objective in Equation 2 is as follows:

$$\arg\max_{\mathcal{G}} \mathcal{F}(\mathcal{G}(\mathbb{Z})) = \sum_{(x,y)\in\mathbb{D}_{\text{test}}} \mathcal{L}(f_{w_p}(x), y) \tag{4}$$

where w_p is the surrogate model's parameters updated with $\mathbb{X}_p$.

We propose an efficient algorithm to iteratively train the generator following three steps: (1) We use the generator $\mathcal{G}$ to generate a number of poisoning queries $\mathbb{X}_p$. (2) $\mathbb{X}_p$ is temporarily used to update $f_s^*(\cdot)$ to obtain w_p. (3) We update the generator to get closer to the optimization objective in Equation 4. By repeating the above three steps, $f_s^*(\cdot)$ is iteratively attacked, and thus the objective value $\mathcal{F}(\cdot)$ becomes larger. Finally, we stop training until the convergence *i.e.*, $\mathcal{F}(\cdot)$ is reached.

To ensure that the generated queries don't significantly deviate from the historical queries, we build a Variational Auto Encoder (VAE) [1] based anomaly detector $\mathcal{D}$ to counterbalance the generator. Specifically, we use some historical queries in the database to train the anomaly detector according to a reconstruction loss $\mathcal{L}_d$. After training, a query will be deemed abnormal if its reconstruction error, as detected by $\mathcal{D}$, exceeds a certain threshold. Then, every time the poisoning queries are generated in the training phase, $\mathcal{D}$ will be triggered to detect abnormal queries among them. Finally, to prevent generating abnormal queries, the generator $\mathcal{G}$ is updated based on the reconstruction loss $\mathcal{L}_d$ associated with these abnormal queries. More details will be introduced in Section 6.

3.4 Attacking

Given a batch of Gaussian noise $\mathbb{Z}$, the trained poisoning query generator $\mathcal{G}$ will output a batch of poisoning queries $\mathbb{X}_p$. Then we can run those queries in the target database. Afterward, the cardinality estimator $f_{w_b}(\cdot)$ in the database will use these queries and their true cardinalities to update itself. Eventually, the cardinality estimator could be poisoned and may not be able to accurately estimate the given testing workload.

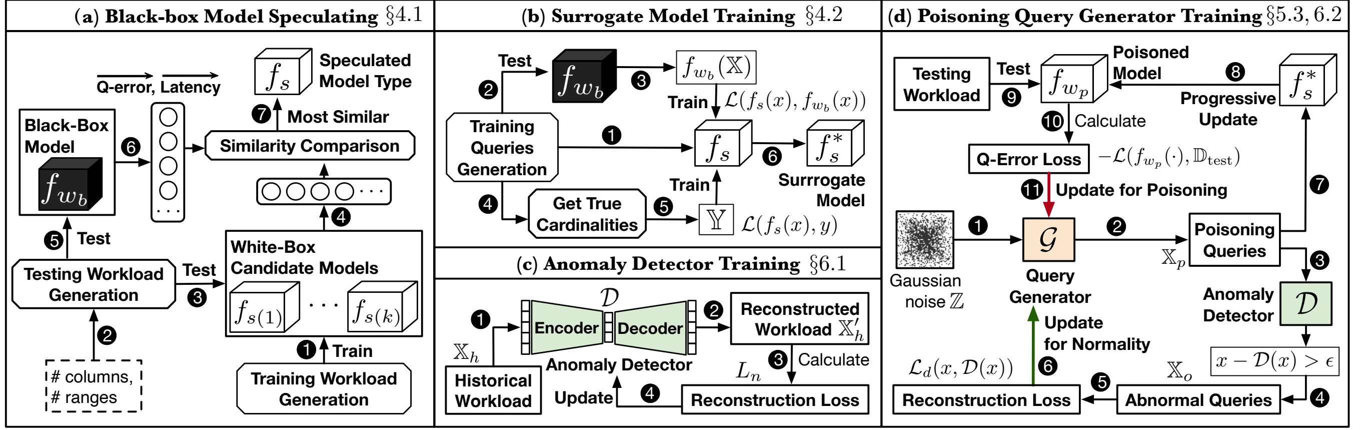

Figure 3: Training workflow of PACE.

4 SURROGATE CE MODEL ACQUISITION

First, we introduce how to speculate the model type in Section 4.1. Next, in Section 4.2, we introduce the strategy of training a white-box model to substitute the black-box model.

4.1 Model Type Speculating of the Black Box

Key idea. Before training a model, it is usually necessary to determine the model type. The reason is that the performance could be quite different even for a same task when using different model types. In our case, we aim to train a white-box model to replace a black-box model. To ensure that they perform similarly, it is essential for them to have the same model type. Therefore, the first step in acquiring a surrogate model is to speculate the type of the black-box model. We compare the performance similarity between the black-box model and the candidate models on a test workload with a specific distribution, and select the model type of the most similar candidate model.

To this end, we first assume k different model types as candidates. Then we propose to pick one from the k types as the type of $f_{w_b}(\cdot)$. At a high level, we pick the type comparing the performance of the k models with that of $f_{w_b}(\cdot)$, based on a set $\mathbb{Q}$ of generated queries. Intuitively, given these queries, if one of the k models performs the most similarly to $f_{w_b}(\cdot)$, they are likely to be with the same type. But note that queries in $\mathbb{Q}$ should have diverse properties (*e.g.*, the column number), to test the performance variations across different model types.

Because when queries have different column numbers and predicate range sizes, the estimated accuracy and latency are much different on different model types. As proposed in [40, 46], (i) the accuracy of MSCN decreases less than FCN when the column number increases. (ii) the accuracy of FCN will be lower than other types when the range of filter predicates is too large or too small, (iii) the inference latency of RNN will increase as the number of columns increases.

Specifically, we first assume the k models with different types as candidates, each of which is trained by a batch of randomly generated training queries.

Second, considering the diverse property discussed above, we generate n_t test queries by varying the number of columns and the range size of filter predicates in queries. Third, we test the k candidate models $f_1(\cdot), \cdots, f_k(\cdot)$ and the black-box model $f_{w_b}(\cdot)$ over these queries, and then compute the mean of Q-error and estimation latency of the $k+1$ models for n_t test queries, producing $k+1$ vectors $\vec{s}_1, \cdots, \vec{s}_k, \vec{s}_b$ with $2 \times n_t$ dimensions. Finally, we calculate the cosine similarity between $\vec{s}_1, \cdots, \vec{s}_k$ and $\vec{s}_b$, and pick the type with the highest similarity as the speculated type. That is, the speculated type is the same as the model type of $f_{i^*}(\cdot)$, where i^* conforms to the following equation:

$$i^* = \arg\max_i \ Cosine(\vec{s}_i, \vec{s}_b) = \frac{\vec{s}_i \cdot \vec{s}_b}{||\vec{s}_i|| \times ||\vec{s}_b||} \quad (5)$$

4.2 Training Strategy

Key idea. After determining the model type, the next step is to train the model parameters. The objective of this task is to enable the white-box model to perform as closely as possible to the black-box model, which requires both models to predict similar outputs for any given input. We utilize not only the estimated cardinalities of the black-box model, but also the ground-truth cardinalities as supervisory information in training the surrogate model. In this way, the surrogate model can achieve strong generalization performance in imitating the black-box model.

A natural approach to training the surrogate model is to use the output of the black-box model as a supervisory information. This can be achieved by initializing the surrogate model with the speculated model type, and generating a batch of training queries $\mathbb{X}$.

The surrogate model $f_s(\cdot)$ can then be trained by imitating the output of the black-box model $f_{w_b}(x)$ on $\mathbb{X}$, where the training loss is defined as follows:

$$\mathcal{L}_s(\mathbb{X}) = \sum_{x \in \mathbb{X}} \mathcal{L}(f_s(x), f_{w_b}(x)) \quad (6)$$

Considering that the black-box model $f_{w_b}(\cdot)$ is trained with true cardinalities in the database as supervision, so $f_{w_b}(\cdot)$ contains the information of true cardinalities. By incorporating the ground-truth labels $\mathbb{Y}$ of $\mathbb{X}$, the surrogate model can learn to better capture the underlying relationships between queries and their true cardinalities, leading to improved generalization performance in imitating

$f_{w_b}(x)$. To this end, we propose a training method that can utilize the information not only from $f_{w_b}(x)$ but also $\mathbb{Y}$ and achieve a smaller imitation error. The loss function is as follows:

$$\mathcal{L}_s(\mathbb{X}, \mathbb{Y}) = \sum_{x \in \mathbb{X}, y \in \mathbb{Y}} \left(\mathcal{L}(f_s(x), f_{w_b}(x)) + \mathcal{L}(f_s(x), y) \right) \quad (7)$$

The former term $\mathcal{L}(f_s(x), f_{w_b}(x))$ can make $f_s(\cdot)$ imitate the $f_{w_b}(\cdot)$ well. And the latter item $\mathcal{L}(f_s(x), y)$ enables the $f_s(\cdot)$ generalizable for unseen queries.

Remark. For the hyperparameters of the surrogate model, we design a default set of parameters, and we will analyze the impact of the inconsistency of hyperparameters in Section 7.

5 POISONING QUERY GENERATION

We first present a high level idea of the training process of the poisoning query generator in Section 5.1. Afterward, we describe the query representation process and the structure of our poisoning query generator in Section 5.2. Finally, we propose an efficient algorithm for training the poisoning query generator in Section 5.3.

5.1 High Level Idea

In order to obtain the diverse poisoning queries, we employ a generator to learn the distribution of poisoning queries and subsequently generate them. As is common in generative networks [8, 32], we provide the generator with Gaussian-distributed noise as input, enhancing its ability to output a variety of queries. Essentially, we enable the generator to learn and transform this Gaussian distribution into the distribution of poisoning queries. Two crucial aspects merit attention. Firstly, to enable the generation of poisoning queries with diverse join patterns, we design a join predicate generator, which creates valid join patterns, then pass them to the predicate generator as a part of input. Secondly, we train the generator using the estimation error of the attacked surrogate model, which serves as the overall objective function.

As shown in Figure 3(d), during each step of the training process of the generator, we feed Gaussian noise to the generator, which outputs poisoning queries (❶-❷). These poisoning queries are used to update the surrogate model (❼-❽), and we use maximizing the estimation error of the updated surrogate model as the objective function (❾-⓫). Since the whole process from the generation of poisoning queries by the generator to the updating of the white-box alternative model is derivable, we use a gradient descent method to update the generator.

5.2 Generator Design

For poisoning queries, we leverage a neural network-based generator to generate them. In this part, we first introduce how we represent a query. After that, we give the detailed structure of the poisoning query generator.

Query Representation. Because most of the current learned CE methods only support SPJ queries, we focus on the generation of SPJ poisoning queries. Formally, consider a database with n tables $\{T_1, ..., T_i, ..., T_n\}$ and m attributes $\{A_1^1, ..., A_i^j, ..., A_n^m\}$. Since the cardinality of a SQL query Q can be determined by two parts, namely the join predicate $J = \bowtie \{T_i\}$ and selection conditions $S = \sigma\{lb_i^j < A_i^j < ub_i^j\}$. Where lb_i^j and ub_i^j represent the normalized upper and lower bounds of the filtering predicate on the attribute A_i^j. The representation process for a query $Q = (J, S)$ into a vector x involves several steps. First, the join predicate J undergoes binary encoding to produce n-dimensional x_{join}, which consists of 0 and 1, where 1 means that the corresponding table lies in J, and 0 otherwise. Second, the selection condition S is encoded into a vector x_{sel} with a dimension of $2\times m$, containing the normalized upper and lower bounds $(lb_1^1, ub_1^1, \cdots, lb_n^m, ub_n^m)$ of the filtering predicates corresponding to the m attributes. In cases where S does not contain a certain attribute A_i^j, the corresponding upper and lower bounds $[lb_i^j, ub_i^j]$ should be [0,1]. Finally, the representation result x is obtained by concatenating x_{join} and x_{sel}.

Generator Structure. To guarantee the diversity (i.e., considering various join combinations of tables) and correctness (i.e., ensuring that the upper bound is greater than the lower bound for each filter predicate) of the generated poisoning queries, we design a generator with three sub-generators based on deep neural networks.

Figure 4 depicts the process of generating a poisoning query, which is composed of 3 generators: the join predicate generator $\mathcal{G}_j$, the lower bound generator $\mathcal{G}_l$, and the range size generator $\mathcal{G}_r$. The $\mathcal{G}_j$ sub-generator is responsible for generating various feasible join predicates for the poisoning queries to ensure the diversity. The $\mathcal{G}_l$ and $\mathcal{G}_r$ sub-generators are combined to generate the lower and upper bounds of the predicates of poisoning queries (the upper bound equals the lower bound plus the range size) according to the join predicates provided by $\mathcal{G}_j$ to ensure the correctness.

Design of $\mathcal{G}_j$. To ensure the diversity of the joins, we feed a Gaussian noise z into $\mathcal{G}_j$, and it outputs a vector x'_{join} of length n, indicating which tables are in the join predicate. The last layer of $\mathcal{G}_j$ is set as a sigmoid activation layer [10] to restrict the values in x'_{join} between 0 and 1, and the value greater than 0.5 indicates that the corresponding table is in the join predicate, otherwise, it is not. After that, to ensure the correctness of the join predicates, we check whether the join predicate represented by x'_{join} conforms to the join schema of the target dataset. If it is not satisfied, the Gaussian noise is regenerated and a new x'_{join} is output. Otherwise, values greater than 0.5 in x'_{join} are set to 1 and the rest are set to 0 to obtain the binary vector x_{join}. To enhance the ability of $\mathcal{G}_j$ to capture correct join predicates, we construct a cross-entropy loss function $\mathcal{L}_j$ to train $\mathcal{G}_j$:

$$\mathcal{L}_j(x'_{join}, x_{join}) = -\sum_{i=1}^{n} x'_{join}[i] \; log(x_{join}[i]) \quad (8)$$

Design of $(\mathcal{G}_l, \mathcal{G}_r)$. To generate predicates, a combination of Gaussian noise z and binary join vector x_{join} is taken as the input of $\mathcal{G}_l$ and $\mathcal{G}_r$, allowing for the generation of diverse predicate upper and lower bounds with the specific join predicate. To guarantee the correctness of the generated predicates, the upper and lower bounds of the predicates for each attribute are not directly generated, as this could result in invalid queries with lower bounds greater than the upper bounds. Instead, $\mathcal{G}_l$ generates the lower bounds of the predicates, and $\mathcal{G}_r$ generates range size. The final layer of both $\mathcal{G}_l$ and $\mathcal{G}_r$ utilizes a sigmoid activation function to normalize the lower bounds and range size of the predicates between 0 and 1. Then we can ensure that the upper bounds are greater than the lower bounds because the upper bounds are calculated by adding

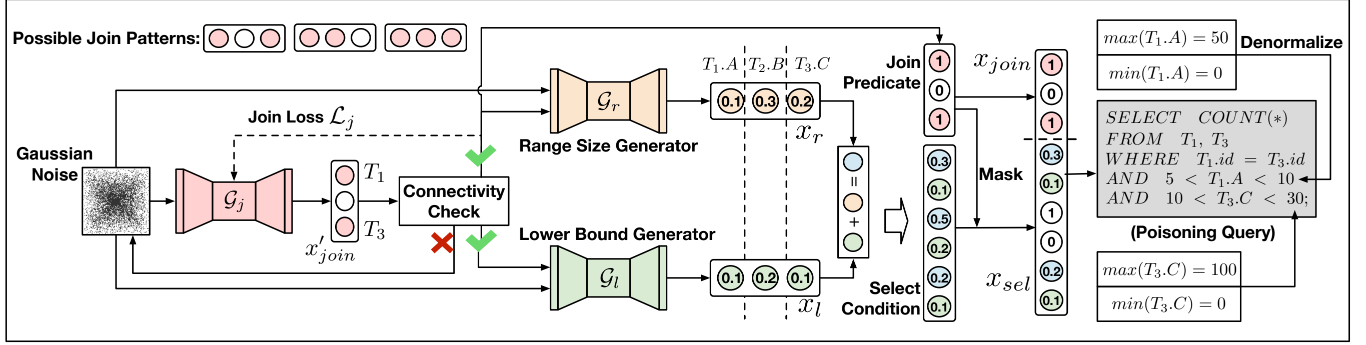

Figure 4: Process of generating a poisoning query. (§5.2)

the lower bounds and range size. Then the x_{sel} can be obtained by a masking process according to x_{join}. That is, if a table is not in the x_{join}, then the lower and upper bounds of the corresponding attributes are set to 0 and 1.

Finally, we can concatenate x_{join} and x_{sel} to obtain the representation x of a poisoning query Q. And this representation can be easily transformed into a query according to the process of $x \rightarrow Q$.

Summarization. $\mathcal{G}_j$ transforms the Gaussian distribution into the correct join predicates. $\mathcal{G}_l$ and $\mathcal{G}_r$ transform the Gaussian distribution into selection predicates with poisoning effectiveness.

5.3 Generator Training

In this part, we discuss the methodology of training the generator. At a high level, the generator produces a batch of poisoning queries, which are utilized to update the surrogate model, and then use the estimation error of the updated model to guide the training of the generator (*i.e.*, solving Equation 4). We first analyze the updating process of the surrogate model to determine its poisoned parameters. Next, we specify our objective function and propose a basic algorithm to train the generator. Due to the high time complexity of the algorithm, we finally propose an algorithm to improve efficiency.

CE model updating. The surrogate model retains existing parameters w_b initially, updates itself on the poisoning queries for a small number (K) of iterations, and ultimately updates its parameters to w_p^K. The update process for one iteration can be formulated as:

$$w_p^e = w_p^{e-1} - \alpha \nabla_{w_p^{e-1}} \sum_{(x,y)\in(\mathcal{G}(\mathbb{Z}),\mathbb{Y}_p)} \mathcal{L}\left(f_{w_p^{e-1}}(x), y\right), \quad w_p^0 = w_b \tag{9}$$

Where w_p^e denotes parameter at the $e-$th iteration during the update process, and $e \in [0, K]$. α represents the learning rate. $w_p^0 = w_b$ means the parameters of the black-box model are initially w_b.

Objective function. Once the parameters of the black-box model have been updated to w_p^K after the K-step update process, our goal is to maximize the objective function $\mathcal{F}$ by optimizing the parameters of the generator $\mathcal{G} = \mathcal{G}_j, \mathcal{G}_r, \mathcal{G}_l$:

$$\underset{\mathcal{G}=\{\mathcal{G}_j,\mathcal{G}_r,\mathcal{G}_l\}}{\arg\max} \; \mathcal{F}(\mathcal{G}(\mathbb{Z}), \mathbb{Y}_p, w_p^K) = \sum_{(x,y)\in\mathbb{D}_{\text{test}}} \mathcal{L}\left(f_{w_p^K}(x), y\right) \tag{10}$$

Since w_p^K varies with $\mathcal{G}$ as described in Equation 9, it is non-trivial to solve this optimization problem.

Lemma 1 (Bivariate optimization). *The problem of poisoning query generation is a bivariate optimization problem that includes two variables, query generator $\mathcal{G}$ and poisoned model w_p. Particularly, w_p is changing with $\mathcal{G}$ when maximizing the objective function.*

Analysis. In the optimization objective as represented in Equation 10, our goal is to maximize the function value $\mathcal{F}$ by optimizing $\mathcal{G}$. However, due to the updating process of the CE model on generated queries, w_p is changing with $\mathcal{G}$ according to Equation 9. Therefore, the objective function must take into account the changing of w_p when optimizing $\mathcal{G}$.

Convergence analysis. Generally, a non-convex optimization problem is guaranteed to converge only if its objective function has the property of Lipschitz continuous gradient [2]. However, since this optimization objective includes a generative neural network and an update process of a CE model, the difficulty of expressing and deducing the large volume of parameters with mathematical formulas poses a notable challenge to prove the convergence from a mathematical perspective. In practice, such problems can typically be optimized to converge using the gradient descent method; namely, we can continually calculate the gradient of $\mathcal{G}$ for $\mathcal{F}$, and modulate $\mathcal{G}$ one step according to the gradient. In addition, to prevent the objective function from converging into a local optimum, we have used large steps in the case of small gradients for escaping from the local optimum. From the perspective of experimental verification, we will report the convergence curves of our optimization problem in Section 7.9.

Basic algorithm of solving equation 10. As shown in Figure 5 (a), a feasible solution is proposed as follows:

<u>(1)</u> Initialize $\mathcal{G}$ to $\mathcal{G}_0$, generate poisoning queries, and ❶ then update the parameters of the surrogate model to w_p^K.

<u>(2)</u> Treat w_p^K in the objective function as constants, and ❷ update $\mathcal{G}_0$ to $\mathcal{G}^M$ through the strategy of gradient descent until the objective function converges for the current w_p^K. We assume that this update process takes M steps.

<u>(3)</u> Obtain new poisoning queries by current $\mathcal{G}^M$. Then ❸ initialize w_p as w_p^0, and ❹ obtain new w_p^K by going through the process as shown in Equation 9 for K steps.

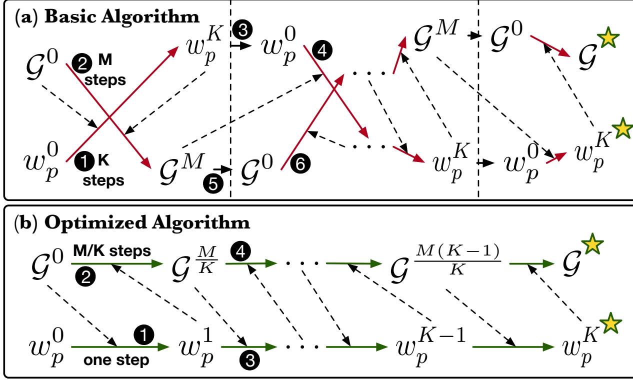

Figure 5: Analysis of the generator training. (§5.3)

(4) Repeat (2) and (3) until the objective function converges.

However, the shortcoming of this solution is that the algorithm complexity is high.

Assuming that (2) and (3) need to be repeated n_r ($n_r > 1$) times, then at each iteration of (2) and (3), w_p is updated for K steps under the current $\mathcal{G}$ instead of the optimal $\mathcal{G}$, and $\mathcal{G}$ is also updated by M steps under the current w_p^K instead of the poisoned w_p^K by optimal $\mathcal{G}$. This leads to that a large number of updates of $\mathcal{G}$ and w_p during each iteration are unnecessary. For example, in the beginning, the w_p update updates K steps for the initialized $\mathcal{G}$, and $\mathcal{G}$ updates M steps according to current w_p^K, in which these M steps are under the misleading guidance. In summary, the basic algorithm updates the generator and the surrogate model separately, which may result in unnecessary updates that do not improve the effectiveness of the poisoning queries.

Acceleration algorithm. To overcome the shortcoming, we propose an efficient algorithm to train the poisoning query generator. At a high level, we can reduce unnecessary update steps by having $\mathcal{G}$ and w_p interact in time. As shown in Figure 5 (b) and Algorithm 1 (now we can ignore the anomaly detector $\mathcal{D}$ and lines 13-15, which will be introduced in Section 6), we can ❷ update the generator with a few steps after each ❶ update of the surrogate model. Specifically, for the inner loop shown in **lines 4-19**, we repeat the following processes (1-7) for l/K times:

(1) Input Gaussian noise $\mathbb{Z}$ to join predicate generator $\mathcal{G}_j$ to obtain the vector set $\mathbb{X}'_j$ and the binary vector set $\mathbb{X}_j$ of join predicates.
(2) Update the $\mathcal{G}_j$ according to the joining loss L_j.
(3) Input $\mathbb{Z}$ and $\mathbb{X}_j$ to lower bound generator $\mathcal{G}_l$ and range size generator $\mathcal{G}_r$ to obtain the selection vector set $\mathbb{X}_s$.
(4) Mask the predicates value of attributes that are not in the join predicates $\mathbb{X}_j$ in $\mathbb{X}_s$ to 0, and concatenate the $\mathbb{X}_j$ and $\mathbb{X}_s$ to get the vector set of generated poisoning queries $\mathbb{X}_p$.
(5) Obtain the cardinality labels $\mathbb{Y}$ of $\mathbb{X}_p$.
(6) Update the surrogate model one step to $f_{\text{tmp}}(\cdot)$ on the poisoning queries (without updating the surrogate model itself).
(7) Take the estimation error of $f_{\text{tmp}}(\cdot)$ on $\mathbb{D}_{\text{test}}$ as the loss L_p, and update the generators $\mathcal{G}_l$ and $\mathcal{G}_r$ by one step according to L_p.

And for each outer loop, we sample the Gaussian noise $\mathbb{Z}$ (**Line 2**) and assign the $f_{\text{tmp}}(\cdot)$ to the surrogate model (**Line 20**).

Algorithm 1: Poisoning Query Generator Training §5.3,6.2

Input: Trained Surrogate CE Model $f_s^*(\cdot)$, trained Anomaly Detector $\mathcal{D}$, number of epochs for CE model updating K and generator training M.
Output: Trained poisoning query generator $\mathcal{G}$.

1 **for** o *in [1, K]* **do**
2 $\quad$ $\mathbb{Z} = \mathcal{N}(0,1)$; // sample noise as generator input
3 $\quad$ **for** i *in* $[\,1+\frac{M(o-1)}{K}, \frac{M\times o}{K}\,]$ **do**
4 $\quad\quad$ $\mathbb{X}'_j = \mathcal{G}_j(\mathbb{Z})$; // join predicate vectors
5 $\quad\quad$ **if** *Check*$(\mathbb{X}'_j)$ == *False* **then**
$\quad\quad\quad$ // if join predicates not conform to schema
6 $\quad\quad\quad$ Continue; // regenerate $\mathbb{X}'_j$
7 $\quad\quad$ $\mathbb{X}_j = \text{Round}(\mathbb{X}'_j)$; // round the values to 0 or 1
8 $\quad\quad$ $L_j = \sum_{(x'_{join}\in\mathbb{X}'_j, x_{join}\in\mathbb{X}_j)} \mathcal{L}_j(x'_{join}, x_{join})$;
9 $\quad\quad$ $\mathcal{G}_j \leftarrow \mathcal{G}_j - \eta\nabla_{\mathcal{G}_j} L_j$;
10 $\quad\quad$ $\mathbb{X}_s = \mathcal{G}_l(\mathbb{Z}, \mathbb{X}_j) + \mathcal{G}_r(\mathbb{Z}, \mathbb{X}_j)$; // selection condition
11 $\quad\quad$ $\mathbb{X}_s = \text{Mask}(\mathbb{X}_p, \mathbb{X}_j)$; // mask $\mathbb{X}_p$ according to $\mathbb{X}_j$
12 $\quad\quad$ $\mathbb{X}_p = \mathbb{X}_j \oplus \mathbb{X}_s$; // poisoning queries
13 $\quad\quad$ $\mathbb{X}_o = \mathbb{X}_p[|\mathbb{X}_p - \mathcal{D}(\mathbb{X}_p)| > \epsilon]$; // abnormal queries
14 $\quad\quad$ $L_n = \mathcal{L}_d(\mathbb{X}_o)$; // reconstruction loss
15 $\quad\quad$ $\mathcal{G}_l \leftarrow \mathcal{G}_l - \eta\nabla_{\mathcal{G}_l} L_n$; $\quad \mathcal{G}_r \leftarrow \mathcal{G}_r - \eta\nabla_{\mathcal{G}_r} L_n$;
16 $\quad\quad$ $\mathbb{Y}_p = \text{Query}(\mathbb{X}_p)$; // get cardinalities of $\mathbb{X}_p$
17 $\quad\quad$ $f_{\text{tmp}}(\cdot) \leftarrow f_s^*(\cdot) - \alpha\nabla_{f_s^*(\cdot)}\mathcal{L}(f_s^*(\mathbb{X}_p), \mathbb{Y}_p)$;
18 $\quad\quad$ $L_p = -\mathcal{L}(f_{\text{tmp}}(\cdot), \mathbb{D}_{\text{test}})$; // estimation error of temporarily updated surrogate model
19 $\quad\quad$ $\mathcal{G}_l \leftarrow \mathcal{G}_l - \eta\nabla_{\mathcal{G}_l} L_p$; $\quad \mathcal{G}_r \leftarrow \mathcal{G}_r - \eta\nabla_{\mathcal{G}_r} L_p$;
20 $\quad$ $f_s^*(\cdot) \leftarrow f_{\text{tmp}}(\cdot)$; // update surrogate model
21 **return** $\mathcal{G}$;

Lemma 2 (Algorithm Complexity). *The time complexity of two generator training algorithms, the basic algorithm and acceleration algorithm, is $O(n_r * (M+K))$ and $O(M+K)$, respectively, where M is the number of update steps required for the generator to converge, K represents the update steps of the CE model on poisoning queries, and n_r is the number of iterations to alternately update the generator and CE model so that the objective function converges.*

Analysis. For the basic algorithm, it contains n_r update processes of the generator and the CE model, so its time complexity is $O(n_r * (M+K))$. The acceleration algorithm only contains one update process of the generator and the CE model, so its time complexity is $O(M+K)$.

Summarization. We incrementally update the generator at each step of updating the surrogate model. This makes $\mathcal{G}$ and w_p interact in time to reduce unnecessary update steps.

6 ENSURE DISTRIBUTION CONSISTENCY OF POISONING QUERIES

If the distribution of generated poisoning queries are significantly different from the distribution of historical queries, the database may recognize these queries as abnormal and not use them to update the CE model. To address this problem, we train an anomaly detector using an unsupervised learning method. The anomaly detector is then deployed against the poisoning query generator.

6.1 Anomaly Detector Training

Key idea. Typically, anomaly detectors are trained by labeling a batch of data as normal or abnormal, followed by training a classification model using supervised learning techniques. However, in our case, we do not have labeled normal and abnormal queries. Instead, we can obtain a set of historical queries $\mathbb{X}_h$, and if the generated poisoning queries distribution is similar to $\mathbb{X}_h$, these queries will not be considered as abnormal queries. So we can train an anomaly detector in an unsupervised way. During the training process, we use MSE loss to guide the anomaly detector to reconstruct the historical queries. Once completed, any query with a reconstruction error that surpasses a predetermined threshold is classified as abnormal.

Specifically, we utilize a variational auto-encoder (VAE) [1] as the anomaly detector $\mathcal{D}$ to reconstruct $\mathbb{X}_h$:

$$x' = \mathcal{D}(x), \quad x \in \mathbb{X}_h \tag{11}$$

To train $\mathcal{D}$, we employ the Mean Squared Error (MSE) loss function [4] as the reconstruction loss.

$$\mathcal{L}_d(\mathbb{X}_h) = \sum_{x \in \mathbb{X}_h} \frac{(x - x')^2}{|\mathbb{X}_h|} \tag{12}$$

Consequently, the VAE's ability to reconstruct a query depends on how different the query is from the distribution of queries in $\mathbb{X}_h$. The greater the consistency, the better the reconstruction performance of the VAE.

After completing the training, we can assess whether an arbitrary query x is abnormal or normal based on the reconstruction error $|x - x'|$. If the error is greater than a predetermined threshold ϵ (e.g., $\epsilon = 0.1$ but $|x - x'| = 0.15$), we classify the query as abnormal.

6.2 Confrontation with Generator

Key idea. We hope to adjust the parameters of the generator so that the poisoning queries $\mathbb{X}_p$ are not easily classified as abnormal queries by the anomaly detector. In essence, we strive to reduce the reconstruction error of the $\mathbb{X}_p$. $\mathbb{X}_p$ is generated by the generator, so the reconstruction loss of $\mathbb{X}_p$ can be backpropagated to the generator, so that we can update the parameters of the generator through the gradient descent method, thereby enhancing the ability of the generator to generate normal queries. During each training iteration of the poisoning query generator, we detect abnormal queries based on the anomaly detector and then use the reconstruction loss to update the generator. This ensures the normality of the $\mathbb{X}_p$ without significantly reducing the poisoning effect.

As shown in Figure 3(c), in order to prevent the generator from generating abnormal queries, we can employ the anomaly detector $\mathcal{D}$ to identify abnormal queries $\mathbb{X}_a$ among the generated queries. Next, we can update the generator using the reconstruction loss of $\mathbb{X}_a$, which is $\mathcal{L}_d(\mathbb{X}_a)$.

Algorithm 1 outlines the process for updating the poisoning queries generator $\mathcal{G}$ using the reconstruction loss of abnormal queries in each inner loop. First, we select abnormal queries based on whether the reconstruction error of the generated queries $\mathbb{X}_p$ exceeds the threshold ϵ (**Line 13**). Then, we calculate the reconstruction loss L_n of these abnormal queries (**Line 14**). Finally, we update $\mathcal{G}$ for one step by computing the gradient of the reconstruction loss with respect to the generator (**Line 15**).

Mechanism analysis. After completing the training process for the anomaly detector, as detailed in Section 6.1, the goal of generating normal queries is transformed into minimizing the reconstruction loss of generated queries (see Equation 12). To achieve the goal, we can get the reconstructed result x' by inputting the generated query x into the anomaly detector. Then, because the gradient between x and $\mathcal{G}$ is differentiable, we can modulate the parameters of the generator $\mathcal{G}$ to minimize $(x - x')^2$ according to $\nabla_{\mathcal{G}}(x - x')^2$, i.e., keep updating $\mathcal{G}$ by a small step in the opposite direction of the gradient.

It is important to note that utilizing the anomaly detector against the generator does not significantly reduce the poisoning effect. The rationale is that we are dealing with a dual-objective optimization problem, where we seek to maximize the effect of the poisoning queries while ensuring that their distribution similar to that of the historical queries. Corresponding to algorithm 1, the first update (**Line 15**) guarantees the normality of the generated queries, while the second update (**Line 19**) ensures the poisoning effectiveness.

7 EXPERIMENTS

This section evaluates the poisoning effect of PACE. We mainly explore the following questions:

- (§7.2) Whether or not PACE's attack on the cardinality estimation models is effective? And what are the differences among different types of cardinality estimation models?
- (§7.3) What is the impact of the poisoning effect concerning the end-to-end query execution performance?
- (§7.4) (1) Can PACE accurately speculate the type of the black-box model? (2) What is the impact if the model type of the surrogate model is different from that of the black-box model? (3) How much better is our method that trains a surrogate model than directly training from the input and output of the black-box model? (4) What is the impact if the hyperparameters of the surrogate model and the black-box model are inconsistent? (5) How does the number of poisoning queries $|\mathbb{X}_p|$ influence the effectiveness of the attack?
- (§7.5) What is the total overhead associated with PACE, including training time, generation time, and attack time?
- (§7.6) How well does our poisoning query generation algorithm perform in terms of effectiveness and efficiency?
- (§7.7) How impactful are the poisoning queries when used on an incrementally trained CE model?
- (§7.8) Can the anomaly detector help generate the normality distribution while preserving the effectiveness of poisoning queries?
- (§7.9) What is the real-world convergence performance of PACE?

Table 2: Query-driven CE models and their hyperparameters.

Model	#Heads	#Layers	HiddenDim	OutDim
FCN [6]	1	4	64×128	1
FCN-Pool [17]	1×3	4	64×128	1
MSCN [19]	3×3	4	64×128	1
RNN [36]	1	4	64	1
LSTM	1	4	64	1
Linear	1	2	128	1

7.1 Experimental Setup

Datasets. We conduct experiments on 4 widely used datasets: **(1)** DMV [35] is a real-world single-table dataset that contains vehicle registration information in New York. **(2)** IMDB [22] is a movie rating dataset that consists of 21 tables.

(3) TPC-H [42] is a popular benchmark dataset that contains 8 tables.

(4) STATS [13] dataset from the Stack Exchange network. Since current query-driven cardinality estimation models fall short of learning string-type data, we encode the string-type attributes into numeric types using dictionaries.

Workloads.

For DMV and TPC-H datasets, we generate 10000 unseen training queries and 1000 testing queries similar to [23, 51, 52] for each CE model. For IMDB and STATS datasets, we generate 10000 unseen training queries and 1000 testing queries based on the templates in IMDB-JOB [22] and STATA-CEB [13] respectively.

CE models. To verify the effectiveness of PACE, we utilize all current neural network-based, query-driven Cardinality Estimation (CE) models, a total of six: **(1)** FCN [6, 17]. A lightweight fully connected neural network. **(2)** FCN+Pool [17]. A neural network that integrates 3 fully connected neural networks with a pooling layer. **(3)** MSCN [19]. A multi-set convolution network. **(4)** RNN [36]. A recurrent neural network. **(5)** LSTM. A long short-term memory network [39]. **(6)** Linear. A simple Linear regression network. The default hyperparameters of the 6 CE models as shown in Table 2.

Baselines. We compare PACE with the performance of the CE models before any attacks (Clean). We compare four baselines of crafting poisoning queries as follows:

(1) Random Generation (Random). Randomly generate a set of queries as described in *Workload* as poisoning queries.

(2) Loss-based Selection (Lb-S). Randomly generate a set of queries and select 10% queries that maximize the inference loss $\mathcal{L}$ of the unpoisoned surrogate model.

(3) Greedy Search (Greedy). Randomly choose a join pattern from the possible join patterns. Then randomly generate 10 range select conditions for each attribute of all tables in the selected join pattern. Finally, construct a poisoning query by selecting one condition for each attribute, with the aim of maximizing the inference loss $\mathcal{L}$ of the unpoisoned surrogate model.

(4) Loss-based Generation (Lb-G). Use the same generator as PACE, but train with the goal of maximizing the inference loss $\mathcal{L}$ of the unpoisoned surrogate model.

Hyper-parameters. The default number of poisoning queries $|\mathbb{X}_p|$ is 450, which accounts for only 5% of the training queries. The number of layers of $\mathcal{G}_j$, $\mathcal{G}_r$, $\mathcal{G}_l$ and $\mathcal{D}$ are 4, 5, 5, 7. The reconstruction threshold ϵ is 5%. The learning rates α and η are both $5e^{-3}$ and Adam [18] optimizer is applied. The number of iterations K for incremental updates of the CE model is 10. The number of iterations l for generator training is 20. The number of outer loops of the basic algorithm n_r is 20.

Metrics. We use four metrics as described in Section 2.2.

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.

7.2 Decline of the CE Models' Accuracy

Average accuracy. The increase in the average Q-error can reflect the overall attack ability of the poisoning methods. Figure 6, 7, 8 and 9 show the average Q-error of each CE model before (Clean) and after being attacked on DMV, IMDB, TPC-H and STATS datasets. We can find that for FCN, FCN+Pool, MSCN, RNN and LSTM, the order of poisoning effectiveness is PACE > Lb-G > Greedy > Lb-S > Random obviously. On average, PACE outperforms the four baselines 2×, 27×, 55×, 212× respectively. The reason for PACE > Lb-G is that Lb-G only focuses on the inference loss before the model is poisoned, but the accuracy of the poisoned model is directly related to the inference loss after the model is poisoned. The reason for Lb-G > Greedy > Lb-S is that Greedy and Lb-S have no training process, resulting in a limited search space for poisoning queries. In addition, The attack effect of PACE on IMDB, TPC-H and STATS is an order of magnitude higher than that on DMV. For example, on IMDB, TPC-H and STATS, PACE increases the estimation error by an average of 370×, 138×, and 89×, respectively, while on DMV it is 26×. This is because the simplicity of a single-table dataset makes it difficult to find queries that can significantly affect the CE models. And we find that the FCN+Pool and MSCN models perform very similarly on different datasets for each poisoning attack method. This is because the architectures of these two models are very similar [17]. Finally, for the Linear CE model, the effectiveness of all attack methods is not obvious, because the Linear regression model has few parameters, which reduces the fitting ability but improves the robustness of the model.

Percentile accuracy. Percentile accuracy in cardinality estimation tasks is also important, especially high percentile accuracy, which can easily affect the performance of database query optimization [53, 57]. Table 3 and 4 shows the percentile Q-error of each CE model before and after being attacked on different datasets.

We can find that for the high percentile Q-error (>90-th), PACE outperforms the Lb-G, Greedy, Lb-S and Random 2.4×, 73×, 135×, 242× respectively, which is much larger than the average Q-error. This is because the optimization goal of PACE is to find queries with the highest poisoning effectiveness directly.

7.3 Impact on End-to-End Execution Time

Decreased accuracy of the cardinality estimator often leads to degraded end-to-end execution performance of the database. In order to verify the effectiveness of PACE on the end-to-end execution performance of the database, we compare PACE and other baselines on the end-to-end execution time (*E2E latency*) in the database.

Table 5 shows the end-to-end execution time of 20 multi-table join testing queries using each CE model before (Clean) and after

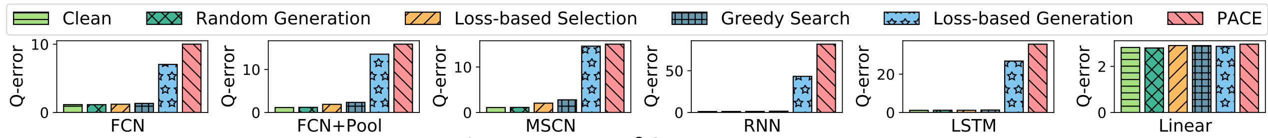

Figure 6: Mean of Q-error on DMV.

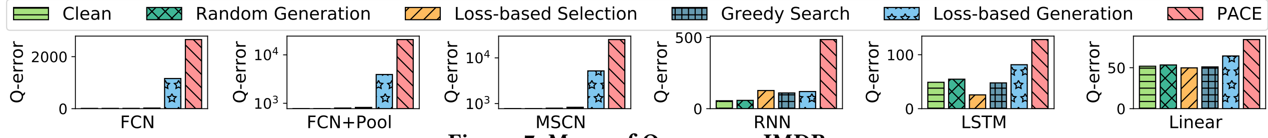

Figure 7: Mean of Q-error on IMDB.

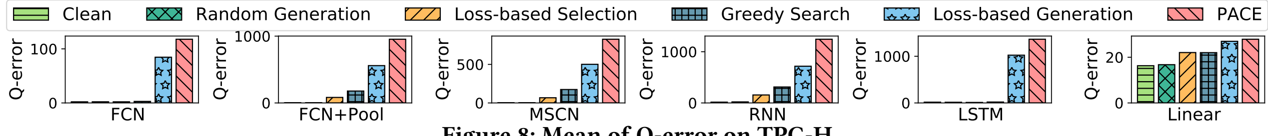

Figure 8: Mean of Q-error on TPC-H.

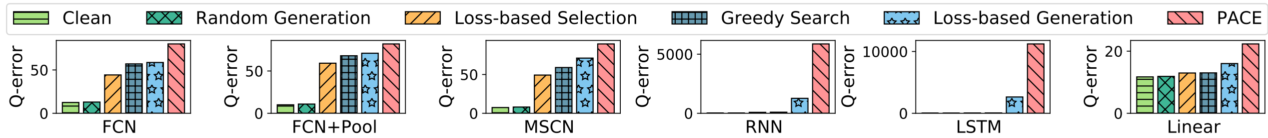

Figure 9: Mean of Q-error on STATS.

Table 3: Percentile Q-errors Results.

		(a) DMV				**(b)** IMDB				**(c)** TPC-H				**(d)** STATS			
CE Model	Method	90th	95th	99th	Max	90th	95th	99th	Max	90th	95th	99th	Max	90th	95th	99th	Max
FCN	Clean	1.263	1.321	1.388	1.420	22.37	51.31	159.6	247.2	3.699	5.787	10.60	13.11	15.57	31.42	99.07	600.2
	Random	1.280	1.330	1.390	1.471	25.68	42.91	165.5	321.5	3.374	6.560	24.47	45.92	15.85	29.35	108.1	615.5
	Lb-S	1.362	1.438	1.633	1.661	33.25	61.99	106.4	202.8	3.685	5.887	13.04	23.64	75.85	143.2	217.2	1682
	Greedy	1.587	1.667	1.837	2.190	72.60	160.8	279.1	617.5	5.300	9.590	24.55	45.78	77.10	155.1	233.2	1699
	Lb-G	23.37	25.55	28.01	31.63	2792	5536	$1.4e^4$	$6.1e^4$	132.8	305.9	1088	5363	79.09	177.8	1032	1887
	PACE	**36.47**	**45.15**	**59.22**	**73.71**	**4581**	**1.4e^4**	**4.1e^4**	**1.3e^5**	**182.1**	**352.8**	**2883**	**6127**	**105.8**	**222.5**	**1261**	**3342**
FCN+ Pool	Clean	1.182	1.138	1.347	1.523	25.73	46.24	123.7	466.3	2.450	2.593	7.995	15.12	11.49	20.25	58.91	310.6
	Random	2.059	1.867	2.293	2.233	70.53	144.7	387.6	1168.7	3.366	4.239	10.85	22.88	20.29	44.42	91.96	223.3
	Lb-S	2.537	2.81	3.515	3.975	392.4	1124	1815	3247	163.6	504.5	1145	2216	62.75	93.07	189.3	414.9
	Greedy	3.287	3.694	4.939	5.372	731.6	2215	3529	7097	271.8	845.1	2146	4914	77.22	114.2	231.6	527.5
	Lb-G	33.67	42.79	59.15	76.22	$1.5e^4$	$3.5e^4$	$8.5e^4$	$3.2e^5$	221.4	5244	$1.4e^4$	$1.5e^4$	79.29	157.1	440.5	945.1
	PACE	**38.16**	**46.57**	**63.60**	**88.22**	**2.2e^4**	**6.2e^4**	**2.3e^5**	**5.7e^5**	**359.4**	**5351**	**1.9e^4**	**2.8e^4**	**90.08**	**172.1**	**871.9**	**2619**
MSCN	Clean	1.173	1.209	1.335	1.359	12.30	31.41	184.2	204.9	2.537	4.547	14.91	20.49	24.51	43.10	109.1	155.8
	Random	1.214	1.253	1.315	1.407	70.09	148.7	335	1223	2.425	4.707	13.95	21.20	29.56	46.43	118.3	135.1
	Lb-S	2.643	2.782	3.402	3.646	258.4	586.6	967.1	2612	97.98	484.0	960.8	2257	193.2	259.8	427.5	1268
	Greedy	3.686	3.683	4.773	4.768	635.4	2190	3431	7555	248.3	1931	3895	$1.0e^4$	257.6	382.2	566.3	2713
	Lb-G	37.75	46.61	63.74	81.02	$1.4e^4$	$3.2e^4$	$9.1e^4$	$2.8e^5$	161.3	4153	$1.0e^4$	$1.3e^4$	2818	3252	7594	9190
	PACE	**38.23**	**47.50**	**64.85**	**83.22**	**2.0e^4**	**6.0e^4**	**2.3e^5**	**4.5e^5**	**485.2**	**4087**	**1.8e^4**	**2.7e^4**	**3622**	**4.5e^4**	**1.3e^5**	**2.2e^5**
RNN	Clean	1.212	1.247	1.328	1.352	134.9	304.0	478.7	1632	37.12	48.12	141.7	179.1	13.09	27.01	72.53	432.2
	Random	1.208	1.275	1.352	1.396	110.4	356.0	555.6	1691	58.71	71.52	85.68	99.81	13.31	34.35	93.53	521.9
	Lb-S	1.430	1.470	1.550	1.590	387.5	846.0	1434	1656	403.1	529.2	775.5	1038	34.01	55.05	158.4	329.4
	Greedy	1.717	1.699	1.823	1.863	346.7	600.7	1217	1349	687.1	828.9	1244	1737	42.01	56.31	198.4	394.9
	Lb-G	57.98	62.75	66.70	72.99	201.6	387.8	793.3	8689	675.6	5598	$1.4e^4$	$2.5e^4$	76.21	149.3	575.9	1156
	PACE	**107.8**	**117.0**	**130.1**	**145.9**	**754.1**	**1664**	**6377**	**4.5e^4**	**1079**	**8437**	**2.6e^4**	**3.3e^4**	**81.57**	**156.8**	**944.5**	**2889**

being attacked on IMDB, TPC-H and STATS datasets. We can find that for each dataset and CE model, PACE achieves the longest end-to-end execution time. From the perspective of execution time increment, PACE outperforms the Lb-G, Greedy, Lb-S and Random (2.5×, 9.6×, 14×, 166×), (2.6×, 3.2×, 3.7×, 119×) and (7×, 24×, 33×, 173×) on IMDB, TPC-H and STATS datasets respectively.

This is because, for multi-table join queries, the accuracy of the cardinality estimation affects the join order and join operator

Table 4: Percentile Q-errors results for LSTM and Linear regression CE models.

		(a) DMV		(b) IMDB		(c) TPC-H	
CE Model	Method	95th	Max	95th	Max	95th	Max
LSTM	Clean	1.262	1.375	202.9	510	47.63	68.81
	Random	1.337	1.527	249.2	668.9	48.16	129.7
	Lb-S	1.333	1.411	275.2	780.5	67.93	212.1
	Greedy	1.496	1.819	315.2	799.9	77.14	204.9
	Lb-G	38.16	43.08	424.2	1168	8867	$3.2e^4$
	PACE	**52.73**	**58.36**	**833.1**	**2275**	**$1.1e^4$**	**$4.0e^4$**
Linear	Clean	5.141	8.128	249.0	1269	53.91	114.9
	Random	5.131	8.046	261.5	1581	62.21	159.4
	Lb-S	5.528	8.724	271.2	1551	80.63	205.7
	Greedy	5.533	8.651	281.2	1873	95.2	218.5
	Lb-G	5.016	8.153	291.3	2061	154.5	494.8
	PACE	5.190	**8.741**	**495.0**	**2258**	**180.1**	471.3

Table 5: End-to-end execution time (s) results.

Dataset	Method	FCN	FCN+Pool	MSCN	RNN	LSTM
IMDB	Clean	560.1	531.7	528	573.8	607.9
	Random	586.4	533.2	528.1	600	632.3
	Lb-S	657.5	712.5	682.6	792.7	691.6
	Greedy	887.1	844.6	796.5	745.0	760.2
	Lb-G	1634	1740	2074	1072	918.1
	PACE	**2412**	**3857**	**4656**	**1214**	**1160**
TPC-H	Clean	61.58	61.92	62.09	65.27	64.49
	Random	62.36	64.68	65.36	68.70	69.10
	Lb-S	87.26	223.2	234.9	96.64	149.7
	Greedy	101.3	237.6	248.0	130.8	164.4
	Lb-G	140.5	262.6	258.7	193.9	198.6
	PACE	**246.8**	**446.6**	**358.6**	**565.5**	**576.4**
STATS	Clean	24.26	23.78	23.47	24.28	24.19
	Random	25.04	24.68	24.48	26.02	25.34
	Lb-S	29.77	29.88	29.87	29.14	30.48
	Greedy	36.80	43.64	39.61	38.70	39.45
	Lb-G	52.41	47.18	51.60	56.83	49.34
	PACE	**190.1**	**180.1**	**185.9**	**282.4**	**247.1**

Table 6: Speculating accuracy for different CE model types.

Dataset \ Black-box	FCN	FCN+Pool	MSCN	RNN	LSTM	Linear
DMV	75%	75%	75%	85%	80%	95%
IMDB	85%	90%	80%	90%	90%	100%
TPC-H	85%	85%	85%	95%	90%	100%
STATS	90%	80%	80%	95%	95%	100%

selection of the query plan, both of which can lead to degradation of the end-to-end execution performance.

7.4 Validation of Surrogate Model

Speculating accuracy. We randomly generate 20 sets of training queries on each dataset to train each type of CE model as a black-box model. Table 6 shows the accuracy of speculating the type of black-box model using our speculating method. We can find that the average accuracy is 87.5%, which illustrates the effectiveness of our speculating method. Among them, the accuracy rate of FCN, FCN+Pool and MSCN is the lowest, which is 82.1% on average. This is because the architectures of the three models are so similar that they are easily speculated as one another.

Table 7: Decrease rates of attack effectiveness when the black-box model type is incorrectly speculated.

Surrogate \ Decrease \ Black-box	FCN	FCN+Pool	MSCN	RNN	LSTM	Linear
FCN	0%	1.74%	5.6%	6.75%	3.85%	3.35%
FCN+Pool	2.66%	0%	0.55%	3.98%	2.36%	3.90%
MSCN	6.80%	0.53%	0%	2.76%	1.53%	4.31%
RNN	29.2%	8.80%	7.68%	0%	2.86%	2.00%
LSTM	13.1%	4.14%	1.59%	2.34%	0%	6.36%
Linear	15.4%	1.83%	1.75%	32.0%	19.7%	0%

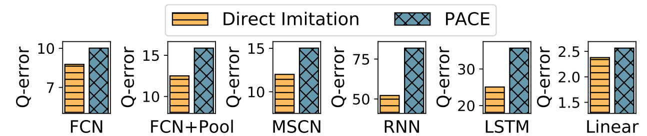

Figure 10: Comparison of our imitation strategy and the direct imitation method.

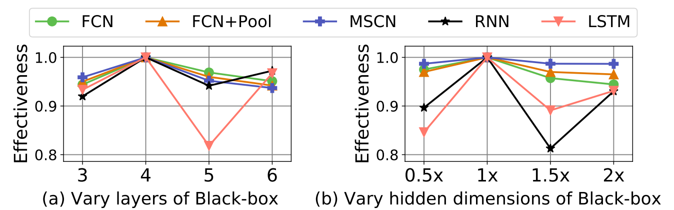

Figure 11: The Attack effectiveness when the hyperparameters of the black-box model change but the surrogate model maintains the default parameters.

Incorrect speculation. To study the influence of the type of black-box model being speculated incorrectly on the attack effectiveness. We train a black-box model for each CE model on DMV dataset, and employ different types of models as surrogate models to study the rate of decline in attack effectiveness of PACE. Table 7 shows the decrease rates of attack effectiveness when the black-box model type is incorrectly speculated. We can find that the average decrease rate is 8.2%, which indicates that overall even if the type of black-box model is incorrectly speculated, the decrease in the attack effectiveness is little.

Effectiveness of our training strategy. To verify the superiority of our imitation strategy as in Equation 7 over the imitation method as in Equation 6 (Direct Imitation), we compare the attack effectiveness of PACE and using Direct Imitation on DMV dataset. As shown in Figure 10, we can find that the attack effectiveness of PACE is on average 32.3% higher than using Direct Imitation, which indicates that our imitation strategy is more effective.

Table 8: The multiplier by which Q-error increases when varying the number of poisoning queries.

Error Increase \ $\lvert\mathbb{X}_p\rvert$ / Dataset	225	450 (default)	900	1800
DMV	5.042×	8.712×	9.233×	9.904×
IMDB	130.9×	241.6×	262.3×	265.9×

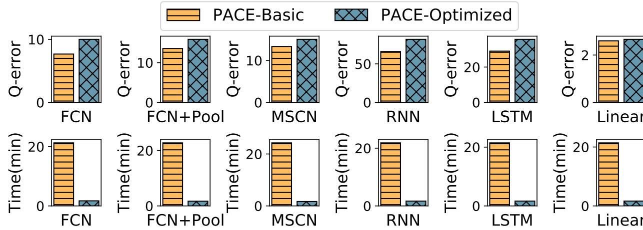

Figure 12: Ablation of the algorithm optimization of PACE.

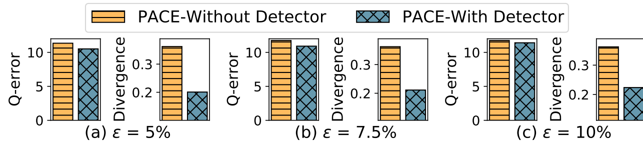

Figure 13: Ablation of using the anomaly detector.

Inconsistent hyperparameters. To study the impact of the inconsistency of hyperparameters of the black-box model and the surrogate model. We train black-box models with different layers and hidden dimensions for each CE model on IMDB dataset, and maintain the default hyperparameters shown in Table 2 for the surrogate model. Figure 11 shows the attack effectiveness of PACE with the change of the black-box model hyperparameters. Where 1.0 on the vertical axis indicates the effectiveness when the hyperparameters of the surrogate model are the same as the black-box model. The horizontal axis in Figure 11(a) represents the number of layers of the black-box model. The horizontal axis in Figure 11(b) indicates the scale of hidden layers' dimension of the black-box model compared to the default hyperparameter. We can find that the average reduction rates are 5.5% and 6.5% for hyperparameters layers and hidden dimensions. This illustrates that the inconsistency of hyperparameters between the surrogate model and the black-box model has a small effect in general.

Varying the number of poisoning queries. We employ a varying number of poisoning queries $|\mathbb{X}_p|$, to attack the FCN model. Table 8 shows the multiples of Q-error increase, relative to the pre-attack model, under the different numbers of $|\mathbb{X}_p|$. We can find that the desired level of attack efficacy is reached with as few as 450 poisoning queries, which is merely 5% of the original training queries. Additional poisoning queries introduced beyond this threshold do not contribute noticeably to the improvement of the attack effectiveness.

Table 9: Overhead evaluation of PACE on different datasets.

Time (s) / Dataset	Training	Generation	Attacking
DMV	188.88	0.5103	1.6069
IMDB	1711.0	0.5343	1.6355
TPCH	823.38	0.5014	1.7063
STATS	3719.8	0.5240	1.6232

Table 10: Overhead evaluation of PACE on different numbers of generated queries.

Time (s) / Number	Training	Generation	Attacking
225 queries	188.88	0.2700	0.861
450 queries	188.88	0.5103	1.6069
900 queries	188.88	1.017	3.124

7.5 Overhead Evaluation

We conduct an experiment to evaluate the overhead of poisoning attacks, including the training time of PACE, the generation time of poisoning queries, and the attacking time, i.e., the updating time of the target cardinality estimation model. Table 9 provides the experiment results of PACE's overhead on FCN across four datasets. The results indicate that the training time of PACE is shortest on the DMV dataset because training on a single-table dataset eliminates the need to train the join predicate generator. The generation time of 450 queries is between 0.5s and 0.55s, and the attacking time of 450 queries is between 1.5s and 1.75s. Table 10 shows the overhead of PACE's under different numbers of poisoning queries on DMV. We find that under different numbers of generated queries, the training time will not change, but the generation and attacking time will proportionally change with the generated queries' number. The reason is that the ratio of the number of generated queries to the batch size determines the generation and attacking time. In summary, the attacking overhead is small.

7.6 Ablation of the Efficiency Optimization

In this section, we will compare PACE before and after algorithm optimization PACE-basic (See Figure 5(a)) and PACE-optimized (See Figure 5(b)) from the perspectives of effectiveness and efficiency.

We explore the effectiveness and efficiency of PACE-basic and PACE-optimized respectively, on the DMV dataset. As shown in Figure 12, we find that on average, PACE-optimized is 20.6% more effective than PACE-basic in terms of attack effectiveness and 9.7 × faster in efficiency. The reason for the improvement in effectiveness is that PACE-basic updates the generator and the surrogate model separately, which results in unnecessary updates that do not improve the effectiveness of the poisoning queries.

7.7 Incrementally Training and Attacking

We conduct an experiment to explore the effect of PACE under an incrementally trained CE model. Specifically, the 10000 training queries described in Section 7.1 are equally divided into five parts

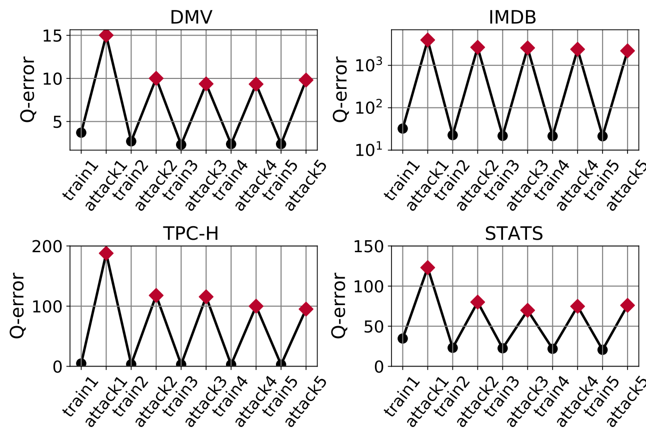

Figure 14: Incrementally training and attacking. The numbers after "train" or "attack" under the x-axis refer to the times of incremental training or attacking.

on four datasets respectively. Then, we train the FCN model incrementally with the divided training subsets. After each time of incremental training, we evaluated the poisoning effectiveness of PACE using the testing queries described in Section 7.1. The experiment results are shown in Figure 14, which illustrates that the Q-error of the first time training and attacking are higher than the following ones. That is because, in the early stages of training on a limited number of queries, the CE model had not yet sufficiently captured the relationships between queries and cardinalities. In the following attacks, our system, on average, increases the Q-error of the CE model by an average factor of 22.4× after each round of incremental CE model training. These experimental results demonstrate the effectiveness and stability of our system.

7.8 Effect of the Anomaly Detector

We determine the abnormality of poisoning queries by comparing its distribution divergence with historical queries.

As shown in Figure 13, we vary the reconstruction error threshold ϵ from 5% to 10%, and compare the effectiveness and normality of poisoning queries of PACE with and without the anomaly detector on DMV dataset. We note the two methods as PACE-Without Detector and PACE-With Detector. We find that PACE-With Detector has a 7.6% decrease in attack effectiveness compared to PACE-Without Detector, but it decreases the abnormality of poisoning query by 72%. That is, using the anomaly detector against the generator does not significantly reduce the poisoning effectiveness of the poisoning queries but ensures the poisoning queries follow similar distribution to the historical workload successfully.

This is because we are solving a dual-objective optimization problem, which is to increase the poisoning effectiveness of the poisoning queries and at the same time ensure the poisoning queries normality. Moreover, the smaller ϵ is, the less divergence is, but the poisoning effectiveness is worse. We recommend choosing 5% for ϵ when using PACE because it maximizes the ratio of the Q-error of the poisoned model and the divergence between poisoning and history queries.

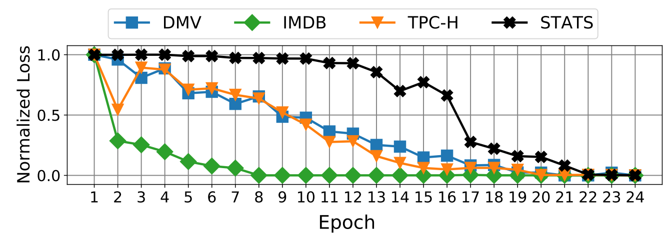

Figure 15: Convergence curve of the optimization objective. The value has been normalized to [0,1].

7.9 Convergence of the Optimization Objective

In order to verify the feasibility of convergence of the optimization problem in Equation 10, we report the changing of the loss function value of the optimization objective in the optimization process on FCN on four datasets. The results are shown in Figure 15. We can find that despite occasional fluctuations, the overall trend continues to decline and converges ultimately.

8 CONCLUSION AND FUTURE WORK

In this work, we study a new problem of poisoning attack on learned cardinality estimation in a black-box setting, and propose a poisoning attack system, PACE.

We devise a method to replace the black-box model with a white-box surrogate model. Then, we design an algorithm to efficiently train a generator that can craft effective poisoning queries. To ensure the poisoning queries follow a similar distribution to historical workload, we propose an anomaly detector against the generator. Experiments show that PACE can efficiently and significantly reduce the accuracy of the CE models.

There are two potential directions for further investigation.

Improve the learned database systems. There are three ways to improve learned database systems directly based on PACE. (1) We can train a classifier to detect abnormal queries by using poisoning queries generated by PACE as training data, and then the classifier can help the learned database systems avoid the attack from poisoning queries. (2) We can test the vulnerability of various cardinality estimation models and recommend a robust one for the learned database systems. (3) As PACE can be adapted to attack other learned regression models, we can apply it to other learned components [11, 12, 16, 26, 27, 41, 59, 60] and improve their security.

Extend to a budget-constrained setting. Typically, the attacker has a limited budget for an attack. Thus there should be a budget that constrains the number of the poisoning queries. One possible approach is to adjust the corresponding parameter of PACE to generate fewer queries. But in order to maximize the poisoning effect of a limited number of poisoning queries, we should design a penalty function, which represents a penalty on the optimization objective when the constraints are not met. This approach allows the solution of the unconstrained problem to converge towards the solution of the constrained problem.

ACKNOWLEDGMENTS

This paper was supported by National Key R&D Program of China (2023YFB4503600), NSF of China (61925205, 62232009, 62102215), Science and Technology Research and Development Plan of China Railway (K2022S005), Huawei, CCF-Huawei Populus Grove Challenge Fund (CCF-HuaweiDBC202309), TAL education, and Beijing National Research Center for Information Science and Technology (BNRist).

REFERENCES

[1] Jinwon An and Sungzoon Cho. 2015. Variational autoencoder based anomaly detection using reconstruction probability. *Special Lecture on IE* 2, 1 (2015), 1–18.
[2] Larry Armijo. 1966. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific Journal of mathematics* 16, 1 (1966), 1–3.
[3] Marco Barreno, Blaine Nelson, Anthony D Joseph, and J Doug Tygar. 2010. The security of machine learning. *Machine Learning* 81, 2 (2010), 121–148.
[4] Peter J Bickel and Kjell A Doksum. 2015. *Mathematical statistics: basic ideas and selected topics, volumes I-II package.* Chapman and Hall/CRC.
[5] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning Attacks against Support Vector Machines. In *Proceedings of the 29th International Coference on International Conference on Machine Learning* (Edinburgh, Scotland) *(ICML'12)*. Omnipress, Madison, WI, USA, 1467–1474.
[6] Anshuman Dutt, Chi Wang, Azade Nazi, Srikanth Kandula, Vivek R. Narasayya, and Surajit Chaudhuri. 2019. Selectivity Estimation for Range Predicates using Lightweight Models. *Proc. VLDB Endow.* 12, 9 (2019), 1044–1057.
[7] GaussDB. 2021. GaussDB is a cloud-based, distributed relational database. https://www.huaweicloud.com/intl/en-us/product/gaussdb.html
[8] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.
[9] Qingsong Guo, Jiaheng Lu, Chao Zhang, Calvin Sun, and Steven Yuan. 2020. Multi-model data query languages and processing paradigms. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management.* 3505–3506.
[10] Jun Han and Claudio Moraga. 1995. The influence of the sigmoid function parameters on the speed of backpropagation learning. In *From Natural to Artificial Neural Computation: International Workshop on Artificial Neural Networks Malaga-Torremolinos, Spain, June 7–9, 1995 Proceedings 3.* Springer, 195–201.
[11] Yue Han, Chengliang Chai, Jiabin Liu, Guoliang Li, Chuangxian Wei, and Chaoqun Zhan. 2022. Dynamic materialized view management using graph neural network. (2022).
[12] Yue Han, Guoliang Li, Haitao Yuan, and Ji Sun. 2021. An autonomous materialized view management system with deep reinforcement learning. In *2021 IEEE 37th International Conference on Data Engineering (ICDE).* IEEE, 2159–2164.
[13] Yuxing Han, Ziniu Wu, Peizhi Wu, Rong Zhu, Jingyi Yang, Liang Wei Tan, Kai Zeng, Gao Cong, Yanzhao Qin, Andreas Pfadler, Zhengping Qian, Jingren Zhou, Jiangneng Li, and Bin Cui. 2021. Cardinality Estimation in DBMS: A Comprehensive Benchmark Evaluation. *CoRR* abs/2109.05877 (2021).
[14] Jiangpeng He, Runyu Mao, Zeman Shao, and Fengqing Zhu. 2020. Incremental learning in online scenario. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 13926–13935.
[15] Benjamin Hilprecht, Andreas Schmidt, Moritz Kulessa, Alejandro Molina, Kristian Kersting, and Carsten Binnig. 2020. DeepDB: Learn from Data, not from Queries! *Proc. VLDB Endow.* 13, 7 (2020), 992–1005.
[16] Shuai Huang, Yong Wang, and Guoliang Li. 2023. ACR-Tree: Constructing R-Trees Using Deep Reinforcement Learning. In *International Conference on Database Systems for Advanced Applications.* Springer, 80–96.
[17] Kyoungmin Kim, Jisung Jung, In Seo, Wook-Shin Han, Kangwoo Choi, and Jaehyok Chong. 2022. Learned Cardinality Estimation: An In-Depth Study. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22).* Association for Computing Machinery, New York, NY, USA, 1214–1227. https://doi.org/10.1145/3514221.3526154
[18] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).
[19] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter A. Boncz, and Alfons Kemper. 2019. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. In *CIDR 2019.*
[20] Evgenios M Kornaropoulos, Silei Ren, and Roberto Tamassia. 2022. The price of tailoring the index to your data: Poisoning attacks on learned index structures. In *Proceedings of the 2022 International Conference on Management of Data.* 1331–1344.
[21] Meghdad Kurmanji and Peter Triantafillou. 2023. Detect, Distill and Update: Learned DB Systems Facing Out of Distribution Data. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–27.
[22] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.
[23] Beibin Li, Yao Lu, and Srikanth Kandula. 2022. Warper: Efficiently Adapting Learned Cardinality Estimators to Data and Workload Drifts. In *Proceedings of the 2022 International Conference on Management of Data.* 1920–1933.
[24] Guoliang Li, Haowen Dong, and Chao Zhang. 2022. Cloud Databases: New Techniques, Challenges, and Opportunities. *VLDB* 15, 12 (2022), 3758–3761.
[25] Guoliang Li and Chao Zhang. 2022. HTAP databases: What is new and what is next. In *Proceedings of the 2022 International Conference on Management of Data.* 2483–2488.
[26] Guoliang Li and Xuanhe Zhou. 2022. Machine learning for data management: A system view. In *2022 IEEE 38th International Conference on Data Engineering (ICDE).* IEEE, 3198–3201.
[27] Guoliang Li, Xuanhe Zhou, and Lei Cao. 2021. AI meets database: AI4DB and DB4AI. In *Proceedings of the 2021 International Conference on Management of Data.* 2859–2866.
[28] Guoliang Li, Xuanhe Zhou, Ji Sun, Xiang Yu, Yue Han, Lianyuan Jin, Wenbo Li, Tianqing Wang, and Shifu Li. 2021. opengauss: An autonomous database system. *Proceedings of the VLDB Endowment* 14, 12 (2021), 3028–3042.
[29] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, et al. 2020. Elastic machine learning algorithms in amazon sagemaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 731–737.
[30] Christopher Manning and Hinrich Schutze. 1999. *Foundations of statistical natural language processing.* MIT press.
[31] Shike Mei and Xiaojin Zhu. 2015. Using machine teaching to identify optimal training-set attacks on machine learners. In *Twenty-Ninth AAAI Conference on Artificial Intelligence.*
[32] Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).
[33] Guido Moerkotte, Thomas Neumann, and Gabriele Steidl. 2009. Preventing bad plans by bounding the impact of cardinality estimation errors. *Proc. VLDB Endow.* 2, 1 (2009), 982–993.
[34] Blaine Nelson, Marco Barreno, Fuching Jack Chi, Anthony D Joseph, Benjamin IP Rubinstein, Udam Saini, Charles Sutton, J Doug Tygar, and Kai Xia. 2008. Exploiting machine learning to subvert your spam filter. *LEET* 8, 1 (2008), 9.
[35] State of New York. 2020. Vehicle, snowmobile, and boat registrations. catalog.data.gov/dataset/vehicle-snowmobile-and-boat-registrations, [Online; accessed November 12, 2020].
[36] Jennifer Ortiz, Magdalena Balazinska, Johannes Gehrke, and S Sathiya Keerthi. 2019. An empirical analysis of deep learning for cardinality estimation. *arXiv preprint arXiv:1905.06425* (2019).
[37] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).
[38] Gaurav Saxena, Mohammad Rahman, Naresh Chainani, Chunbin Lin, George Caragea, Fahim Chowdhury, Ryan Marcus, Tim Kraska, Ippokratis Pandis, and Balakrishnan Narayanaswamy. 2023. Auto-WLM: Machine learning enhanced workload management in Amazon Redshift. In *Companion of the 2023 International Conference on Management of Data.* 225–237.
[39] Alex Sherstinsky. 2020. Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network. *Physica D: Nonlinear Phenomena* 404 (2020), 132306.
[40] Ji Sun, Jintao Zhang, Zhaoyan Sun, Guoliang Li, and Nan Tang. 2021. Learned cardinality estimation: A design space exploration and a comparative evaluation. *Proc. VLDB Endow.* 15, 1 (2021), 85–97.
[41] Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. 2023. Learned Index: A Comprehensive Experimental Evaluation. *Proceedings of the VLDB Endowment* 16, 8 (2023), 1992–2004.
[42] TPC. 2021. Tpch benchmark. http://www.tpc.org.
[43] Vladimir Vapnik. 1991. Principles of risk minimization for learning theory. *Advances in neural information processing systems* 4 (1991).
[44] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2021. FACE: A Normalizing Flow based Cardinality Estimator. *Proc. VLDB Endow.* 15, 1 (2021), 72–84.
[45] Jiayi Wang, Chengliang Chai, Jiabin Liu, and Guoliang Li. 2023. Cardinality estimation using normalizing flow. *The VLDB Journal* (2023), 1–26.
[46] Xiaoying Wang, Changbo Qu, Weiyuan Wu, Jiannan Wang, and Qingqing Zhou. 2021. Are We Ready for Learned Cardinality Estimation? *Proc. VLDB Endow.* 14, 9 (oct 2021), 1640–1654. https://doi.org/10.14778/3461535.3461552
[47] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.
[48] Ziniu Wu, Amir Shaikhha, Rong Zhu, Kai Zeng, Yuxing Han, and Jingren Zhou. 2020. BayesCard: Revitilizing Bayesian Frameworks for Cardinality Estimation. *arXiv preprint arXiv:2012.14743* (2020).
[49] Han Xiao, Huang Xiao, and Claudia Eckert. 2012. Adversarial label flips attack on support vector machines. In *ECAI 2012.* IOS Press, 870–875.

[50] Chaofei Yang, Qing Wu, Hai Li, and Yiran Chen. 2017. Generative poisoning attack method against neural networks. *arXiv preprint arXiv:1703.01340* (2017).
[51] Zongheng Yang, Amog Kamsetty, Sifei Luan, Eric Liang, Yan Duan, Peter Chen, and Ion Stoica. 2020. NeuroCard: One Cardinality Estimator for All Tables. *Proc. VLDB Endow.* 14, 1 (2020), 61–73.
[52] Zongheng Yang, Eric Liang, Amog Kamsetty, Chenggang Wu, Yan Duan, Xi Chen, Pieter Abbeel, Joseph M. Hellerstein, Sanjay Krishnan, and Ion Stoica. 2019. Deep Unsupervised Cardinality Estimation. *Proc. VLDB Endow.* 13, 3 (2019), 279–292.
[53] Xiang Yu, Chengliang Chai, Guoliang Li, and Jiabin Liu. 2022. Cost-based or learning-based? A hybrid query optimizer for query plan selection. *Proceedings of the VLDB Endowment* 15, 13 (2022), 3924–3936.
[54] Chao Zhang and Jiaheng Lu. 2020. Selectivity estimation for relation-tree joins. In *32nd International Conference on Scientific and Statistical Database Management.* 1–12.
[55] Chao Zhang and Jiaheng Lu. 2021. Holistic evaluation in multi-model databases benchmarking. *Distributed and Parallel Databases* 39 (2021), 1–33.
[56] Chao Zhang, Jiaheng Lu, Pengfei Xu, and Yuxing Chen. 2019. Unibench: A benchmark for multi-model database management systems. In *Performance Evaluation and Benchmarking for the Era of Artificial Intelligence: 10th TPC Technology Conference, TPCTC 2018.* Springer, 7–23.
[57] Jintao Zhang, Chao Zhang, Guoliang Li, and Chengliang Chai. 2023. AutoCE: An Accurate and Efficient Model Advisor for Learned Cardinality Estimation. In *2023 IEEE 39th International Conference on Data Engineering (ICDE).* IEEE, 2621–2633.
[58] Rui Zhang and Quanyan Zhu. 2017. A game-theoretic analysis of label flipping attacks on distributed support vector machines. In *2017 51st Annual Conference on Information Sciences and Systems (CISS).* IEEE, 1–6.
[59] Xuanhe Zhou, Guoliang Li, Chengliang Chai, and Jianhua Feng. 2021. A learned query rewrite system using monte carlo tree search. *Proceedings of the VLDB Endowment* 15, 1 (2021), 46–58.
[60] Xuanhe Zhou, Guoliang Li, Jianhua Feng, Luyang Liu, and Wei Guo. 2023. Grep: A Graph Learning Based Database Partitioning System. *Proceedings of the ACM on Management of Data* 1, 1 (2023), 1–24.

Received July 2023; revised October 2023; accepted November 2023