PIPA: Attacking Learning-based Index Advisors via a Probing-Injecting Framework

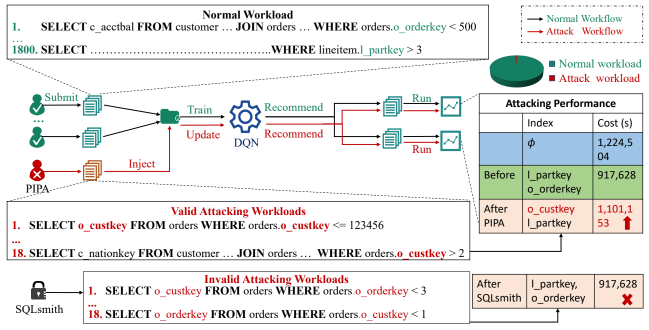
ABSTRACT

Despite the promising performance of recent learning-based Index Advisors (IAs), they are susceptible to malicious attacks. This paper makes the first attempt at poisoning attacks against learning-based IAs, by injecting a maliciously crafted attacking workload to retrain the victim IA so that its indexing performance on normal workloads will be significantly degraded. Poisoning attacks against IAs are under the opaque-box setting, i.e., the victim's internals and the training workloads are unknown. There are three challenges, i.e., *how to probe "index preference" from opaque-box IAs*, *how to design effective attacking strategies even if the IAs can be finetuned*, and *how to generate queries with different constraints for IA probing and attacking*. We present an attacking system `PIPA` with a probing stage, an injecting stage, and a query generator. To address the first challenge, the probing stage estimates the victim IA's action preference on indexable columns by observing its responses to the probing workload. To address the second challenge, the injecting stage injects workloads that spoof the victim IA to demote the top-ranked indexes in the estimated action preference and promote mid-ranked indexes. Thus the attack is effective because the victim is trapped in a local optimum even after fine-tuning. To address the third challenge, `PIPA` utilizes `IABART` (`Index Aware BART`) to generate queries that can be optimized by building indexes on a given set of indexes. Extensive experiments on different benchmarks against various learning-based IAs demonstrate the effectiveness of `PIPA`.

1 INTRODUCTION

Index selection is crucial to the performance of relational database systems [26]. Traditionally, index selection relies on expert database administrators [18] to analyze the workload and data characteristics and select the appropriate set of indexes. To reduce labor expenses, various *heuristics algorithms* [6, 8, 31, 40] have been proposed to guide the search of possible indexes. Since the heuristics are manually designed, their capabilities are usually limited, i.e., they can miss good indexes that are suitable for a particular workload/data. Recently, *learning-based index advisors* [19, 20, 26, 29, 30, 34, 41, 45], which utilize machine learning techniques to tailor index configurations for different workloads and data, have gained numerous attention from both academia and industry.

Although learning-based index advisors have shown promising performance, they are susceptible to malicious attacks. For example,



Normal Workload
1. **SELECT** c_acctbal **FROM** customer … **JOIN** orders … **WHERE** orders.o_orderkey < 500
…
1800. **SELECT** ……………………………………………**WHERE** lineitem.l_partkey > 3

Valid Attacking Workloads
1. **SELECT o_custkey FROM** orders **WHERE** orders.**o_custkey** <= 123456
…
18. **SELECT** c_nationkey **FROM** customer … **JOIN** orders … **WHERE** orders.**o_custkey** > 2

Invalid Attacking Workloads
1. **SELECT** o_custkey **FROM** orders **WHERE** orders.o_orderkey < 3
…
18. **SELECT** o_orderkey **FROM** orders **WHERE** orders.o_custkey < 1

Attacking Performance

	Index	Cost (s)
	ϕ	1,224,504
Before	l_partkey o_orderkey	917,628
After PIPA	o_custkey l_partkey	1,101,153 ⬆
After SQLsmith	l_partkey, o_orderkey	917,628 ✗

Figure 1: Example poisoning attacks against index advisors

Figure 1[1] shows that an index advisor (i.e., DQN [20] based on deep reinforcement learning) leads to performance degradation if its training data is poisoned. Suppose a supplier with multiple franchisees sharing the same cloud business solution. Their data is stored on the same database instance, and their table structures are totally or partially consistent due to using the same solution. Their franchisees and employees submit daily workloads, and most of them rely on the "`l_partkey,o_orderkey`" columns. The database management system collects daily workloads to train DQN [20] and has achieved good indexing performance, i.e., it recommends indexes "`l_partkey,o_orderkey`" and reduces the execution cost by 25%. At some point, an attacker (e.g., a reckless employee or a malicious franchisee) submits a workload accessing the "`o_custkey`" column. DQN updates its parameters by *re-training* itself on both the normal workloads and the attacking workload. Even though the attacking workload is relatively small (i.e., 18 queries in the attacking workload w.r.t 1800 queries in the normal workloads), the performance of DQN is significantly decreased. It recommends sub-optimal index configurations, i.e., "`o_custkey, l_partkey`", to the normal daily workloads, and the execution cost is 20% higher than before.

The above example demonstrates that poisoning attacks against learning-based index advisors (IAs) can have severe consequences. Poisoning attacks have been studied in recommendation systems [12, 14], natural language processing [10, 39], and computer vision [9, 38]. However, to the best of our knowledge, poisoning attacks against learning-based IAs have never been well investigated before. To realize poisoning attacks against IAs in practice, it is important to adopt an "**opaque-box**" setting, i.e., the attacker faces an encapsulated IA whose training details are unavailable. This means that the attacker has no access to the IA's internals, e.g., the algorithms applied, the index candidates considered, the exact values of the model parameters, etc. The attacker has no knowledge of the IA's training data, e.g., the normal workloads. The attacker cannot control the training process, e.g., provide the wrong reward

Conference'17, July 2017, Washington, DC, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

[1]This demonstration is conducted on TPC-H. Details are shown in the supplementary file in https://anonymous.4open.science/r/PIDPA-73DB.

or directly interfere with the index selection. The attacker can only interact with the IA by submitting an input workload and observing the IA's output.

Under above settings, most existing poisoning attack methods are infeasible because they require a certain amount of knowledge of the training process, e.g., the gradient [27], the rewards [1, 24, 28, 37, 44] or action space [23], the labels of the training data [17], etc. In particular, there are three challenges in conducting a poisoning attack on opaque-box IAs.

C1: Victim IA Probing. Obviously, the attack will be ineffective without any information about the victim IA. The problem is, what information can be useful to guide the attack? Furthermore, the only available signals are the victim's output on some *probing workloads*. It remains a difficult problem to design the probing workload to reveal useful information about the victim.

C2: Effective Attack Strategies. The only possible way to poison the training data is to inject attacking workloads. It is a non-trivial problem to design attacking workloads to reduce the performance of victims with diverse design details. Some IAs, such as SWIRL [19], are deployed in an *one-off* fashion, i.e., once the IA is trained or re-trained, it makes direct predictions for any workload. Others [20, 26, 29, 30] are *trial-based*, i.e., after being well-trained, the IA still iterates several times to produce trial indexes for a given workload. For the *trial-based* IAs, the attacking workload only alters their initialization, and the impacts of an attacking strategy are far more complicated.

C3: Query Generation for IA Probing and Attacking. A query generator is needed to automate the attacking procedure. However, there are a large number of valid SQL queries, but only a small number of them can be used to attack IAs. It is hard to identify poisoning SQL queries from a large valid SQL space. Existing query generators are not applicable because their goal is not to identify poisoning SQL queries. For example, SQLsmith [32], which focuses on generating queries that satisfy the syntax constraint, cannot produce valid attacking workloads. As shown in Figure 1, the attack will fail if we use queries generated by SQLsmith.

We study the problem of opaque-box poisoning attacks against IAs and propose `PIPA` (`Probing-Injecting Poisoning Attack`) (Section 2). `PIPA` consists of a *probing stage* before an *injecting stage*, both of which are based on a query generator `IABART` (`Index Aware BART`). Several novel techniques are proposed in `PIPA` to address the challenges.

To address **C1**, the probing stage (Section 4) estimates the victim's *action preference* on all indexable columns (i.e., which column is likely to be chosen to build an index), by observing the victim's output to a few probing workloads. Furthermore, the probing stage is iterative, i.e., the probing workloads are updated based on previous observations to more accurately estimate the action preference within a probing budget (i.e., the number of probing workloads).

To address **C2**, the injecting stage (Section 5) designs attacking workloads that can actively spoof the victim IA to demote the top-ranked indexes in the estimated action preference and promote mid-ranked indexes. Since top-ranked indexes will likely perform well on training workloads, degrading them leads to inferior indexing performance for "one-off" IAs. Furthermore, promoting mid-ranked instead of low-ranked indexes provides an initialization that more effectively traps the victim in a local optimum. Thus, the attacking strategy is also effective for "trial-based" IAs.

To address **C3**, `IABART` targets on generating a SQL query that can be optimized by building an index on some given columns (Section 3). `IABART` is based on the powerful backbone BART [21], which has been widely used in NLP. To enable `IABART` to generate a SQL query from scratch for probing and attacking, we propose a novel progressive training framework to capture the complex correlations among the query, the required index, and the corresponding indexing performance. We also propose a novel decoding method to guarantee that `IABART` can generate syntactically correct queries.

In summary, our contributions are four-fold. (1) We make the first attempt at poisoning attacks against learning-based index advisors. (2) We propose an attacking framework `PIPA`, which can achieve stable attack effects under an opaque-box setting to different IAs. (3) We propose a novel training framework to train Large Language Models (e.g., BART) to generate SQL queries that satisfy certain index-aware requirements. (4) We conduct comprehensive experiments on different benchmarks and demonstrate that various learning-based IAs are vulnerable to `PIPA`.

2 PROBING-INJECTING POISONING ATTACK

In this section, we first introduce the concept of index advisors (IAs) and formally define data poisoning attacks against IAs (Section 2.1). Next, we describe the framework of `PIPA` (Section 2.2).

2.1 Problem Definition

Definition 2.1. (Index Advisor) Given a workload $\mathbb{W}$ on a dataset d, an index advisor $\mathcal{IA}$ aims to build a set of indexes $\mathbb{I}^{w,d} = \mathcal{IA}(\mathbb{W}, d)$, such that the performance improvement of executing the workload $\mathbb{W}$ with indexes $\mathbb{I}^{w,d}$ against without the indexes is maximized, under the budget constraint that the index size $\mathcal{B}(\mathbb{I}^{w,d})$ is not larger than a budget bound B, i.e., $\mathcal{B}(\mathbb{I}^{w,d}) < B$.

We aim to attack index advisor to make it lose its effectiveness in improving performance, by inserting malicious workloads.

During the poisoning attack, the attacker's knowledge and capabilities are limited. First, the IA's internals are invisible to the attacker, and only the IA's output is exposed. Second, the attacker owns data tables that are partially consistent with other tenants (i.e., normal users). Only the data schema of the common tables is visible to the attacker. Third, the attacker cannot observe or control other tenants' workloads, but they can submit a deliberately crafted workload to harm the performance of the IA.

Definition 2.2. (Opaque-box Poisoning Attack against IAs) Consider a victim $\mathcal{IA}$ with an invisible learning module $\mathcal{A}_\theta$, unknown parameters θ trained on a dataset d, and hidden normal workloads $\mathbb{W}^1, \cdots, \mathbb{W}^M$. The attacker injects a maliciously crafted workload $\tilde{\mathbb{W}} = \{\tilde{q}\}$ of N_a attacking queries. The victim IA will be re-trained on the combined set $\{\mathbb{W}^1, \cdots, \mathbb{W}^M, \tilde{\mathbb{W}}\}$ to update the parameters and obtains $\tilde{\mathcal{A}}_{\tilde{\theta}}$. The goal is to harm the IA's performance on the normal workloads, i.e., improving the workload execution cost. $\forall 1 \le m \le M, c(\mathbb{W}^m, d, \tilde{\mathcal{IA}}(\mathbb{W}^m, d)) > c(\mathbb{W}^m, d, \mathcal{IA}(\mathbb{W}^m, d))$.

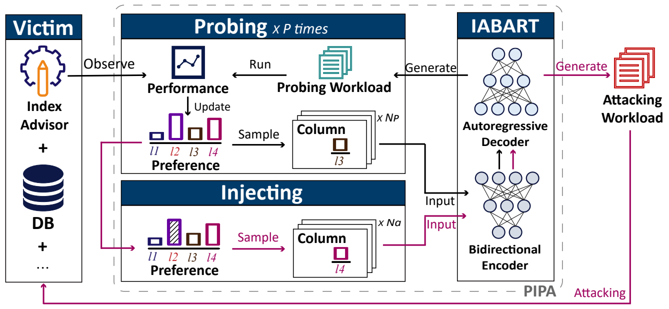


Figure 2: Framework overview of PIPA

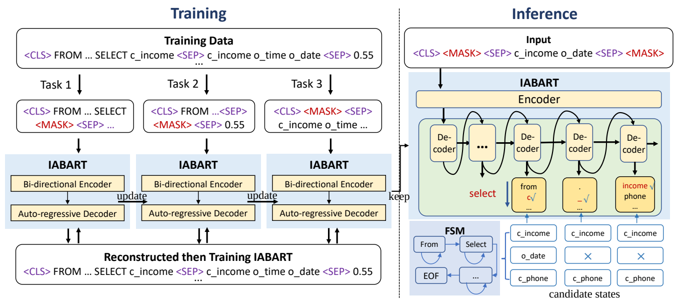


Figure 3: Overview of the training and inference of IABART

2.2 PIPA Overview

Workflow. PIPA (Probing-Injecting Poisoning Attack) is composed of three modules, i.e., query generator IABART, index advisor probing, index advisor injecting, as shown in Figure 2. The training of IABART is independent of the attacking procedure. In each attack, the attacker implements the probing stage before the actual injecting stage, and IABART is called in both stages.

Index Advisor Probing. Since the model parameter (i.e., θ) and training workloads are unknown under the opaque-box setting, we want to extract information related to θ from the victim IA's responses to different probing workloads. Intuitively, the victim's *action preferences*, i.e., which column is more likely to be chosen by the victim regardless of the probing workloads, can reveal critical information about how θ decides indexes. According to the victim IA's training goal, the preferred columns will likely be effective on the training workloads $\mathbb{W}^1, \cdots, \mathbb{W}^M$, so the cost of training workloads is minimized on the victim's chosen indexes.

To avoid costly probing, i.e., extensive index overhead to many/large probing workloads, the probing workloads should be carefully designed to estimate action preference accurately. Inspired by trial-and-error methods such as Bayesian Optimization [33], we resort to an iterative approach (i.e., the number of iterations is less than a probing budget). As shown in Figure 2, in each iteration, (1) a small-sized probing workload is generated by calling the query generator, (2) the probing workload is implemented to observe the victim's performance, (3) the performance is exploited to estimate the action preference, (4) the probing workload is renewed to explore other possibilities of the action preference (e.g., a different ranking of columns). To renew the probing workloads, PIPA samples columns currently ranked in lower positions of the estimated action preference and generate probing workloads that can be optimized by building indexes on these columns. Thus, the victim's responses in the next iteration will likely change the order of columns and refine the current estimate (details in Section 4.3).

Index Advisor Attacking. We can use the action preference to supervise the attack. Specifically, as shown in Figure 2, we can generate attacking workloads that (1) can be optimized by building indexes on less preferred columns and (2) can not be optimized by building indexes on the most preferred indexes. Injecting such attacking workloads and re-training the victim will eventually down-weigh the preferred columns in θ. As a result, the updated parameters $\tilde{\theta}$ will be less effective on normal workloads.

Query Generator. The probing and attacking stages both need a query generator to yield a workload of queries that satisfy specific index-aware performance requirements. The input of IABART is a set of columns, which are specified by the probing or the injecting stage; the output is a query that can be optimized by building indexes on these columns.

3 QUERY GENERATOR

The probing and injecting stages raise a query generation problem to meet index-aware performance requirements. Formally, given a set of n columns $\mathbb{I} = \{l_1, \cdots, l_n\}$, the attacker's own data tables d, the goal is to generate a query q that the optimal index for q is the input column set, i.e., $\forall \mathbb{I}'$ with $|\mathbb{I}'| = n$, $c(q, d, \mathbb{I}) \leq c(q, d, \mathbb{I}')$.

Existing query generators can not fulfill the above goal. For example, in Figure 1, even though the column is fixed as "o_custkey", the SQL generator [32] produces a query that the optimal index is "o_orderkey", which is inconsistent with the input constraint. Our intuition is to train a SQL generator based on the backbone BART [21], which has demonstrated a strong ability to generate sequences in NLP. However, the conventional BART can not be directly applied due to three challenges. (1) The original training of BART is based on masked sequence completion for a natural language sequence, which is infeasible to train a query generator to meet certain index-aware performances. (2) The training task is designed to capture context relationships within the sequences, which is inefficient in generating an entire query sequence given its optimal indexing columns. (3) BART can not guarantee the syntactic correctness of the output query (i.e., executable), which is unacceptable in probing and injecting.

Thus, we present the construction of training data (Section 3.1), a novel progressive training paradigm (Section 3.2), and a syntactically correct inference method (Section 3.3).

As shown in Figure 3, IABART contains a bi-directional encoder and an auto-regressive decoder. The bi-directional encoder reads an input text sequence $\mathbf{x}^{mask}$ corrupted from $\mathbf{x}$ (e.g., some tokens are masked) from both directions (i.e., left to right and right to left) and produces a representation $E(\mathbf{x}^{mask})$. The auto-regressive decoder outputs a sequence $\mathbf{y}$ that recovers the input text from left to right, based on the encoder's representation and previously generated tokens, i.e., $\mathbf{y}_t = D\big(E(\mathbf{x}^{mask}), \mathbf{y}_{<t}\big)$. To utilize the common-sense

knowledge learned from pre-training in large-scale English language corpus, we initialize the model by BART-base[2] with 6-layer Transformer and GeLU activation.

3.1 Construction of Training Data

To construct the ground-truth sequences $\{\mathbf{x}\}$, three parts are concatenated, i.e., the query, the indexes, and the performance. Each $\mathbf{x}$ is in the form of "`<CLS>` q `<SEP>` $\mathbb{I}^{q,d}$ `<SEP>` $\mathcal{R}(\mathbb{I}^{q,d})$", where `<CLS>` is a special token to mark the beginning of a sequence, `<SEP>` segments the sequence, q is a SQL query, d is the attacker's own data, $\mathbb{I}^{q,d}$ is a sequence of indexes, $\mathcal{R}(\mathbb{I}^{q,d})$ is the corresponding indexing performance.

q is generated by feeding a random seed to the Finite State Machine (FSM) [43] on d. We tokenize the query because a SQL query usually contains words that are rare in the open-domain corpus but are important clues to suggest the data structure. For example, "`customer.c_income`" rarely appears in the corpus on which BART-base is trained, so it will be abrupt for BART-base. But its token segments are important in the SQL query because they suggest the table `customer` and the column `c_income`. Therefore, we use the sub-token level tokenizer to segment word `customer.c_income` to five tokens, i.e., `customer`, `.`, `c`, `_`, `income`.

To associate different SQL queries with their appropriate index configurations, we use SWIRL [19] to recommend a set of indexes $\mathbb{I}^{q,d}$ for each query. We use SWIRL because it is a State-Of-The-Art IA with superior indexing performances. Moreover, it can adapt to different workloads and make index advice on the fly, thus reducing the time cost to construct the ground-truth data.

The inclusion of $\mathcal{R}(\mathbb{I}^{q,d})$ in the training sample is to help `IABART` to understand the benefit of $\mathbb{I}^{q,d}$ and further enhance its accuracy in generating a SQL query to meet the index requirements. We compute $\mathcal{R}(\mathbb{I}^{q,d}) = \frac{c(q,d,\emptyset)-c(q,d,\mathbb{I}^{q,d})}{c(q,d,\emptyset)}$, where $c(q,d,\emptyset)$ is the estimated cost of running the query without any index, $c(q,d,\mathbb{I}^{q,d})$ is the estimated cost with the appropriate index. We use estimated cost instead of the actual cost to speed up the construction and collect more training samples. Note that $\mathcal{R}(\mathbb{I}^{q,d})$ is rounded up to two decimal places (e.g., 0.31).

3.2 Progressive Masked Span Prediction

The masked span prediction pre-training task is found to be more suitable in generating and predicting spans of text [16]. Each sequence in the training set is corrupted by masking a sequence span starting from s to e. The masked token is replaced by a special <MASK> symbol to form $\mathbf{x}^{mask}$. Then, the masked span prediction attempts to recover the corrupted sequence by optimizing the following loss function:

$$\mathcal{L}(E,D) = \sum_{t=s}^{t=e} -logp\left(\mathbf{x}_t = D\big(E(\mathbf{x}^{mask}), \mathbf{y}_{<t}\big)\right). \tag{1}$$

However, the masked span prediction task itself is ineffective as our goal is to generate the whole SQL query. We present progressive masked span prediction, which is motivated by the human learning process, i.e., human learning abilities are enhanced by progressively training from the easiest task to the hardest task. Thus, we present three pre-training tasks.

As shown in Figure 3, in the first task, each mini-batch randomly draws a sequence from the training set and corrupts the sequence by randomly masking one token in the sequence. The first task encourages the model to learn correlations among tokens, e.g., to predict a missing token in a SQL query, to predict a missing index given the entire SQL query and other indexes, or to predict the possible indexing performance. It is the easiest task since only one token is missing each time.

The second task strengthens the model's understanding of the association between indexes and a SQL query by masking all indexes $\mathbb{I}^{q,d}$. Note that instead of masking a single token, this task masks a sub-sequence, and hence is more difficult than the first task and encourages the model to improve itself.

The third task masks the whole query sequence q and keeps only the index configuration and required performance. This task is the most difficult. Thus, by training progressively with the three tasks, `IABART` can capture the complex relationships among the SQL query, the index, and the indexing performance. Furthermore, the third task generates a SQL query from scratch, which is close to the inference task in the probing and injecting stages.

3.3 Inference

In the inference, the input is in the form "`<CLS> <MASK> <SEP>` $\mathbb{I}$ `<SEP> <MASK>`", where $\mathbb{I}$ are a set of indexable columns specified in the probing or injecting stage. The trained `IABART` is implemented to fill the masked part to be extracted as a SQL query.

The conventional decoding strategy in inference is greedy search (i.e., in each step, selecting the token with the largest probability) or beam search (i.e., expanding the greedy search and returning a list of most likely sequences). These decoding strategies can not be used because the generated sequence might have incorrect grammar. We present a novel decoding approach based on FSM.

Specifically, in each generation step, given the previously generated tokens, the model will look for the candidate states in FSM and search the decoder in a top-down manner to adopt the first token that matches a candidate state. The dictionary of the decoder is collected from the training samples. Since the FSM is based on words instead of segmented tokens, we adopt prefix matching to adapt to our tokenizer. An example is shown in Figure 3, the previously generated token is "`select`", and the candidate states by the FSM include "`c_income, o_date, c_phone`". The next word is generated by combining several tokens. For the first token, in the decoder's generation list, "`c`" will be selected because "`from`" does not match the prefix of any of the FSM's candidate states. Once "`c`" is selected, '`o_date`" will be deleted from the candidate states, and "`c_income, c_phone`" are reserved for future prefix matching. Next, the decoder updates its output ranking. "`_`" will be selected because "`c_`" matches the prefix of FSM's candidate states. In the third step, "`income`" will be selected, and "`c_phone`" will be deleted from the FSM's candidate states.

[2]https://huggingface.co/facebook/bart-base

4 PROBING INDEX PREFERENCE

4.1 Derivation of Action Preference

Although some victim IA's action space includes multi-column indexes, we only extract information regarding **single-column indexes** in the probing stage for practical reasons.(1) The internal architecture of the victim IA, including the set of multi-column index candidates, is unknown. (2) The enumeration space of possible combinations of columns is too large, leading to inaccurate information given the probing budget. (3) Single-column indexes also reveal valuable information about multi-column indexes. Because the primary column of a multi-column index is accessed first, the indexing performance of a multi-column index is primarily related to the first single-column index.

Therefore, we seek to derive a **ranking** over all indexable columns as the action preference. We use a ranking instead of the actual numeric value for two reasons. First, in the opaque-box setting, the specific value of inner parameters θ is unknown. Second, the ranking order (i.e., which indexable column is preferred over the other) provides substantial supervision in generating the attacking workload.

To determine the order position of each column, our key idea is to measure the "preference" of column l_i by aggregating over different workloads (i.e., to compute the expectation). The intuition is illustrated in Figure 4. Generally, the weight of a victim choosing l_i based on a workload $\mathbb{W}$ is expressed in $\theta(l_i, \mathbf{s}^{\mathbb{W}})$, where $\mathbf{s}^{\mathbb{W}}$ is a variable dependent on the workload. For example, $\mathbf{s}^{\mathbb{W}}$ can be the state variable in most reinforcement learning IAs [19, 20, 26, 29, 30] [3]. As shown in Figure 4(a), $\theta(l, \mathbf{s})$ is visually plotted as a contour on the continuous variable $\mathbf{s}$ and the discrete variable l. Note that the exact value of θ is invisible everywhere to the attacker because the attacker has no access to the parameters of the victim and the training workloads. Nonetheless, calculating the expectation of different states, i.e., $\mathcal{K}(l_i) = \mathbb{E}_{\mathbf{s}}\theta(l_i, \mathbf{s})$ can eliminate the state variable, plotted as the black line in Figure 4(b). The resulting $\mathcal{K}(l_i)$ represents a tendency that the victim IA favors a column regardless of the training workloads.

There are two advantages of using the expectation. First, it allows us to attack without knowing the training details, e.g., the training workload, the index trajectory of a reinforcement learning IA, etc. Instead, it reflects workload-independent features in the training procedure, e.g., the column's selectivity, inherent bias due to the model's masking strategy to prune certain index candidates, etc. Second, we can compute an empirical expectation by aggregating over probing workloads because each probing workload can produce a sample of $\mathbf{s}$.

Furthermore, we weigh each action by the performance reward it achieves. The estimated action preference is based on current parameters, which will be updated in the retraining and make the estimation obsolete. To make the attack effective, we incorporate the benefit of taking each action. It provides more information to guide the attack to strike along directions that significantly impact indexing performance.

[3] Reinforcement learning IAs usually consist of two modules, i.e., a state representation module to encode the current dataset, current action, and current workload into $\mathbf{s}$, and a policy network to output a chosen index by $\mathcal{A}_\theta(l_j|\mathbf{s})$

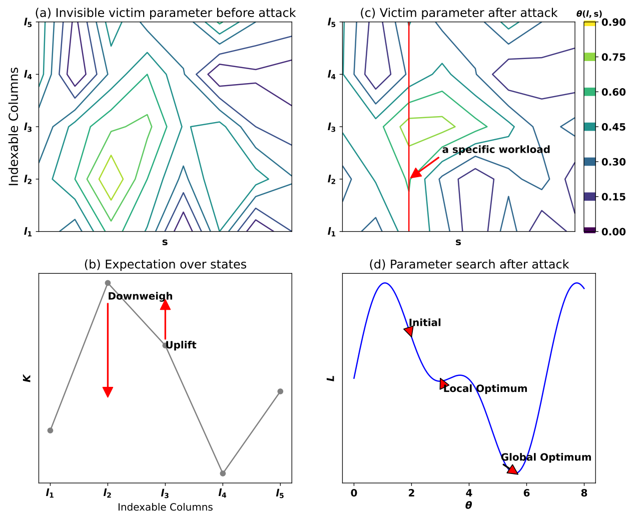


Figure 4: Intuition of probing and attacking

The above reasoning gives the definition of action preference,

$$\begin{aligned} &\mathbf{k} = < l_1, \cdots, l_L >, \\ &\forall i < j, \quad \mathcal{K}(l_i) > \mathcal{K}(l_j), \\ &\mathcal{K}(l_i) = \mathbb{E}_{\mathbf{s}}\left[\mathcal{R}(l_i, \mathbf{s})\theta(l_i, \mathbf{s})\right], \end{aligned} \tag{2}$$

where $\mathbf{k}$ is the action preference, which is a ranking over the set of indexable columns $l_1, \cdots, l_L$. $\mathcal{R}(l_i, \mathbf{s})$ is the "reward" of building an index on column l_i for a particular state $\mathbf{s}$, $\theta(l_i, \mathbf{s})$ is the exact value of the victim's parameter at point $(l_i, \mathbf{s})$, $\mathbb{E}_{\mathbf{s}}$ is the expectation over all possible states.

4.2 Calculation of Action Preference

$\mathcal{K}$ can be computed by the empirical expectation based on the victim IA's feedback to the probing workloads. Suppose the probing stage repeats for P iterations, and each iteration generates a probing workload that contains N_P probing queries. Let's denote a probing workload as $\mathring{\mathbb{W}}^p, 1 \leq p \leq P$, the state related to this probing workload is $\mathbf{s}^p$, and the victim's output index configuration is $\mathbb{I}^p$.

First, we can approximate $\theta(l_i, \mathbf{s}^p)$ in Equation 2. In each inference step t, the victim IA outputs an index with the maximal action probability. It means that if $l_i \in \mathbb{I}^p$, then $\forall l_j \notin \mathbb{I}^p, \theta(l_i, \mathbf{s}^p) > \theta(l_j, \mathbf{s}^p)$. Also, since each column appears in the output indexes at most once, this is equivalent to using a sparse policy $\mathring{\theta}$,

$$\mathring{\theta}(l_i, \mathbf{s}^p) = \begin{cases} 1 & l_i \in \mathbb{I}^p, \\ 0 & l_i \notin \mathbb{I}^p. \end{cases} \tag{3}$$

Next, we can calculate the reward function $\mathcal{R}(l_i, \mathbf{s})$ in Equation 2. We assume each action contributes equally to the long-term reward. The most common long-term reward in IAs [19, 20, 26] is the relative cost reduction. Thus, we define a reward function

$$\mathring{\mathcal{R}}(l_i, \mathbf{s}^p) = \begin{cases} \dfrac{1 - \frac{c(\mathring{\mathbb{W}}^p, d, \mathbb{I}^p)}{c(\mathring{\mathbb{W}}^p, d, \emptyset)}}{|\mathbb{I}^p|}, & \text{if} \quad l_i \in \mathbb{I}^p \\ 0 & \text{else,} \end{cases} \tag{4}$$

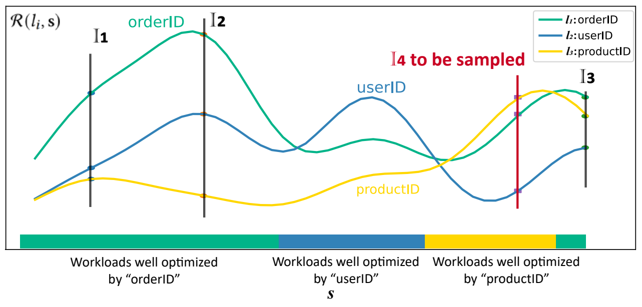


Figure 5: Illustration of the Probing Strategy

where $c(\mathring{\mathbb{W}}^p, d, \mathbb{I}^p)$ is the actual execution cost of the probing workload using the victim IA's recommended index configuration, $\emptyset$ is the null index, $|\mathbb{I}^p|$ is the number of indexes in the output.

Finally, the empirical expectation for $\mathcal{K}$ is computed by

$$\mathcal{K}(l_i) = \frac{1}{P}\sum_{p} \mathring{\mathcal{R}}(l_i, \mathbf{s}^p)\mathring{\theta}(l_i, \mathbf{s}^p) = \frac{1}{P}\sum_{p: l_i \in \mathbb{I}^p} \frac{1 - \frac{c(\mathring{\mathbb{W}}^p, d, \mathbb{I}^p)}{c(\mathring{\mathbb{W}}^p, d, \emptyset)}}{|\mathbb{I}^p|}. \quad (5)$$

4.3 Probing Strategy

Clearly, the computing of $\mathcal{K}$ is affected by probing workloads $\mathring{\mathbb{W}}^1, \cdots, \mathring{\mathbb{W}}^P$. Given the probing budget P, N_P for the number of probing epochs and the size of the probing workloads, it is important to balance exploitation (i.e., using the information already known) and exploration (i.e., gathering new information). Towards this end, we propose a novel probing strategy to generate probing workloads each round that can potentially reveal new information from previous probing rounds.

The intuition is illustrated in Figure 5. Suppose there are three columns l_1, l_2, l_3 = "orderID","userID","productID". The rewards of $\mathring{\mathcal{R}}(l, \mathbf{s})$ are visualized as the three lines in Figure 5 (they are unknown until observed) where the x-axis denotes the $\mathbf{s}$. We highlight three areas in the x-axis, i.e., states produced by workloads that the three indexing columns can optimize. Each probing epoch generates a probing workload, which draws a sample of $\mathbf{s}$ and yields an observation point in the three lines. Based on the first three observations $\mathbb{I}^1, \mathbb{I}^2, \mathbb{I}^3$, the ranking metric $\mathcal{K}(\cdot)$ can be calculated, and the ranking order is "orderID" >"userID" >"productID". In the next round, to collect new information, we prefer to generate a workload that likely changes the previous column order. Intuitively, we can choose column "productID" that ranks at the lowest position, and generate a workload that is likely to be optimized using "productID". In this manner, the victim's output $\mathbb{I}^4$ is likely to include "productID", and the observed reward will promote the ranking of "productID". Note that the fourth observation does not necessarily change the column order, but it will likely reveal more information than using other probing workloads, e.g., workloads that can be optimized by building an index on "orderID".

Motivated by the above intuition, we propose the probing strategy. As shown in Algorithm 1, the input includes the environment (e.g. victim IA, dataset), query generator IABART and hyperparameters such as probing budget P, N_p, number of specified columns $|\{c\}|$ (which will be discussed in Section 6). The probing stage initializes a probability vector over all indexable columns μ to a uniform distribution (line 1), i.e., each column has an equal probability of being sampled. Then, the probing stage repeats the process of maximal P times (line 2), in each time N_p probing queries can be generated (line 3). Note that P and N_p are user-defined probing budgets. A set of column c is sampled based on the current column probability (line 4). The IABART in Section 3 is implemented to generate a probing query to be put in $\mathring{\mathbb{W}}^p$ that can be optimized by building indexes on $\{c\}$ (line 5). Let the victim IA recommend index configuration $\mathbb{I}^p$ for $\mathring{\mathbb{W}}^p$ on dataset d (line 7). For each column in the index configuration $\mathbb{I}^p$, the relative cost reduction is observed to update $\mathcal{K}$ (line 8). The column probability μ is updated accordingly (line 9) for the next round.

Algorithm 1: Probing procedure

```
Data: An opaque-box victim $\mathcal{IA}$, the dataset $d$, the query
      generator IABART, probing budget $P, N_p$, number of
      specified columns $|\{c\}|$
Result: The victim IA's action preference $\mathbf{k}$
1  $\mu^1 \leftarrow \mathcal{U}(0, L)$ ;
2  for $p \leftarrow 1$ to $P$ do
3      for $i \leftarrow 1$ to $N_p$ do
4          $\{c\} \sim \mu^p$;
5          $\mathring{\mathbb{W}}^p \leftarrow \mathring{\mathbb{W}}^p \cup IABART(\{c\})$;
6      end
7      $\mathbb{I}^p \leftarrow \mathcal{IA}(\mathring{\mathbb{W}}^p, d)$ ;
8      Update $\mathcal{K}()$ by Equation 5;
9      Update $\mu^{p+1}$ by Equation 6;
10 end
11 Return $\mathbf{k}$
```

To update the probability of sampling columns, we compute:

$$\begin{aligned} \bar{\mu}(l_j)^p &= |min(\mu(l_j)^{p-1} - \alpha\frac{1}{p-1}\sum_{i<p}\mathring{\mathcal{R}}(l_j, \mathbf{s}^i) - \beta, 0)|, \\ \mu(l_j)^p &= \frac{\bar{\mu}(l_j)^p}{\sum_j \bar{\mu}(l_j)^p}. \end{aligned} \quad (6)$$

We explain the details of Equation 6, where $\mu(l_j)^p$ is the probability of sampling l_j in round p. A column's probability will be decreased if it receives a higher rank in previous iterations, i.e., larger $\sum_{i<p}\mathring{\mathcal{R}}(l_j, \mathbf{s}^i)$, α is a coefficient to control the trade-off between exploitation and exploration.

β is a parameter to avoid exploring unwanted index columns. For example, suppose a column l_j with low index selectivity cannot serve as an index. In that case, the IA will never recommend it, thus receiving a zero reward score $\mathring{\mathcal{R}}(l_j, \mathbf{s}^i)$ in every previous round i. $\mu(l_j)$ will be too large that l_j will be drawn to generate the next probing workload. But since the probing workload can not be optimized by building an index on l_j, the next probing is non-informative. Thus, in Equation 6, if the reward of a particular column l_j is rarely observed in all previous rounds and thus $\mu(l_j)^{p-1} - \alpha\frac{1}{p-1}\sum_{i<p}\mathring{\mathcal{R}}(l_j, \mathbf{s}^i) > \beta$, then we use the $min(\cdot, 0)$ function to force $\mu(l_j) = 0$.

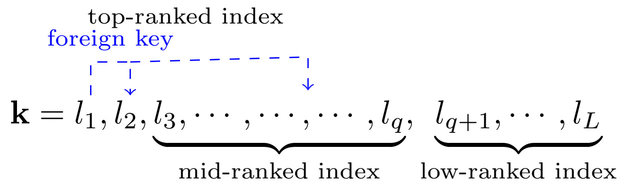

top-ranked index
foreign key
$\mathbf{k} = l_1, l_2, l_3, \cdots, \cdots, \cdots, l_q, \; l_{q+1}, \cdots, l_L$
mid-ranked index
low-ranked index

Figure 6: Illustration of index segments

Finally, we use the absolute function $|\cdot|$ and normalization to guarantee that μ^p is a probability function.

5 INJECTING

Based on the action preference $\mathbf{k}$ detected from the probing stage, we can generate the attacking workload $\tilde{\mathbb{W}}$. We only inject the attacking workload once, assuming the victim IA will re-train on the new training set after injection.

As shown in Figure 6, $\mathbf{k}$ can be grouped into three segments: top-ranked indexes l_1, mid-ranked indexes $l_2, \cdots, l_q$, and low-ranked indexes $l_{q+1}, \cdots, l_L$, where the division boundaries are controlled by hyper-parameter q. In addition, we treat the best index and its foreign keys as the top-ranked index for reasons that we will discuss in Section 6.4.

The intuition is that $\tilde{\mathbb{W}}$ can be optimized by building indexes on columns that fall into a *target segment*. Thus, during the re-training, the victim IA will be encouraged to promote columns in the target segment and demote columns elsewhere.

Clearly, attacking workloads should not contain queries that are optimized by top-ranked indexes. The top-ranked columns are likely to perform well on the training workloads. The attack will be invalid if the attacking workloads strengthen the top-ranked columns.

We argue that the target segment can not be low-ranked indexes for the following reasons. (1) Some columns in $l_{q+1}, \cdots, l_L$ are bad indexes, e.g., they have low index selectivity. Therefore, if we use them as the input for IABART, the generated queries will be more likely non-sargable, i.e., queries that can not be optimized by indexing. These queries will yield a reward close to zero no matter what the victim IA changes its index configuration. Thus, injecting these attacking queries will have little impact on re-training the IA. (2) The low-ranked columns usually do not appear frequently in the victim workload, meaning their attacking effects are not generalized to different IAs. For example, if we use a column that never appeared in the training workload, it will never be considered as an index candidate by SWIRL [19].

Therefore, we choose the mid-ranked indexes as the target segment. As shown in Figure 4(b), our goal is to *downweigh* the best column (e.g., l_2 in the figure) and *uplift* the mid-ranked column (e.g., l_3). As illustrated in Figure 4(c), the most preferred column is changed from l_2 to l_3 after attacking. Given a normal workload (red vertical line in the figure), the victim will likely select l_3, which is sub-optimal. Moreover, operating on mid-ranked indexes is efficient for both one-off and trial-based victims. For one-off victims, we already illustrate in Figure 4(c) that the changed model parameter will lead to degraded performance. For trial-based victims, as shown in Figure 4(d), the attack will give a bad initialization in the victim's search procedure to minimize its loss function. The victim is more likely to be trapped in the local optimum. More experimental discussions are presented in Section 6.2.

We sample the target columns for each query in the attacking workload and generate a query by IABART. We use sampling instead of defining a fixed set of target columns to increase the diversity of the generated queries. Having more diverse queries in the attacking workload has the following advantages: (1) the column coverage of the attacking workload is broader, which helps the attacking workload to bypass some indexing candidate filtering heuristics; (2) diverse queries help to improve the invisibility of the attack, i.e., attacking queries assessing a fixed set of columns are easier to be detected.

Algorithm 2: Attacking procedure

```
Input: IABART, estimated action preference
       $\mathbf{k} = \{l_1, \cdots, l_N\}$, the dataset $d$, boundary $q$, number
       of columns $|\{c\}|$, attacking workload size $N_a$.
Result: The attacking workload $\tilde{\mathbb{W}}$
1 for $i \leftarrow 1$ to $N_a$ do
2     $\{c\} \sim \{l_2, \cdots, l_q\}$;
3     $\tilde{q} \leftarrow IABART(\{c\})$;
4     if $c(\tilde{q}, d, \{c\}) < c(\tilde{q}, d, l_1)$ then
5         $\tilde{\mathbb{W}} \leftarrow \tilde{\mathbb{W}} \cup \tilde{q}$;
6     end
7 end
8 Return $\tilde{\mathbb{W}}$
```

Motivated by the above intuition, we propose the attacking procedure. As shown in Algorithm 2, the input includes query generator IABART, estimated action preference $\mathbf{k}$ and hyper-parameters such as mid-ranked index boundary q, number of specified columns $|\{c\}|$, attacking workload size N_a. N_a attacking queries are generated (line 1). For each query, a set of columns $\{c\}$ is randomly sampled from the mid-ranked indexes interval controlled by hyper-parameters q (line 2). The IABART in Section 3 is implemented to generate an attacking query (line 3). The generated query is filtered to ensure that the top-ranked index is not the optimal index (line 4). Thus, the attacking workload merges queries that (1) can be optimized by building indexes on mid-ranked columns while (2) can not be optimized by indexing on top-ranked columns (line 5).

6 EXPERIMENT

In this section, we first verify the attack performance of PIPA against existing learning-based IAs on different benchmarks (Section 6.2). We then investigate the impact of hyper-parameters, including the size of the attacking workload (Section 6.3), the division boundaries in the attacking strategies (Section 6.4), and the number of probing epochs (Section 6.5). Finally, we evaluate the performance of IABART (Section 6.6).

6.1 Experimental Setup

Datasets. Since most learning-based IAs only support analytic workloads, the experiments are conducted on analytic benchmarks:

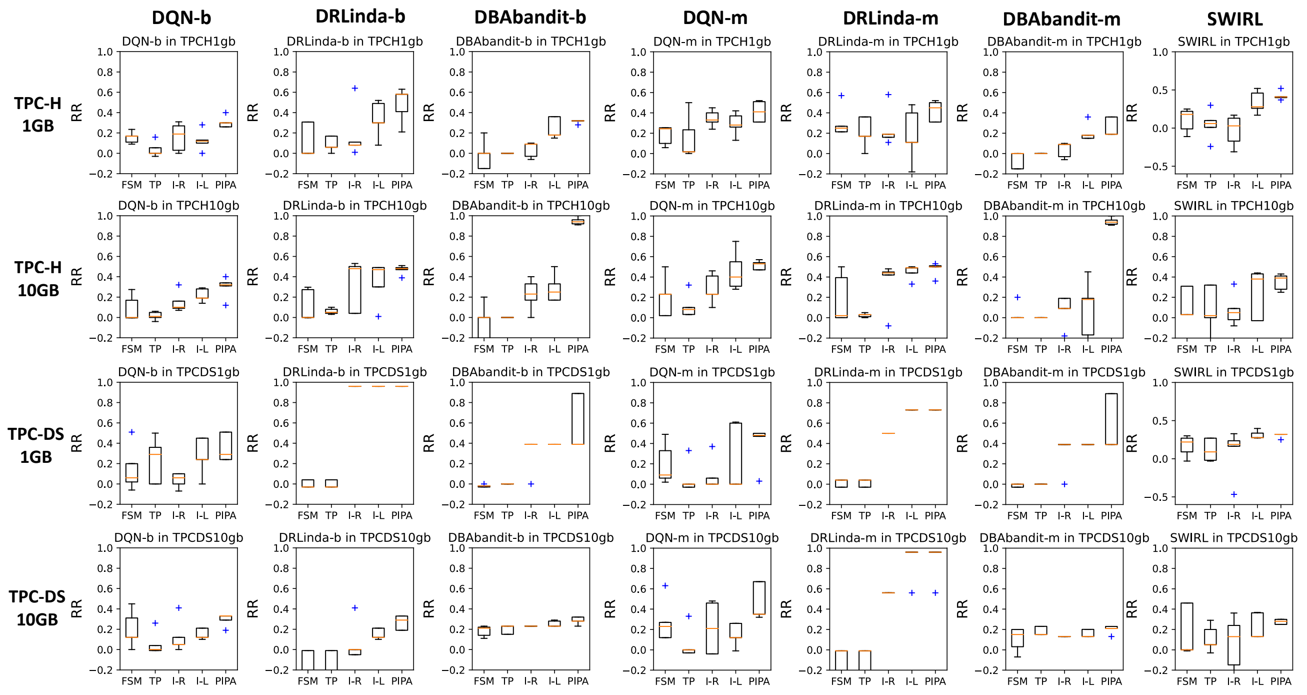


Figure 7: Attacking performance in terms of Reward Reduction (RR) by different methods against various victims

TPC-H[4] and TPC-DS[5]. For each benchmark, two different data sizes are generated, i.e., 1GB and 10GB.

Victim Index Advisors. We select four learning-based index advisors, i.e., DBAbandit [26] models index selection as a multi-armed bandit problem; DQN [20] and DRLindex [29, 30] adopt the Deep Q-Network algorithm; and SWIRL [19] adopts the proximal policy optimization algorithm. These victims cover typical learning-based IA design paradigms, except for the Monte Carlo Tree Search (MCTS) methods. We do not attack MCTS models in this paper because they do not have the ability to update after workload changes.

To train and retrain these victim IAs, 400 trajectories[6] (20 for DBAbandit because it converges fast) are produced for each workload. For inference, we let DQN and DRLindex produce 400 trajectories (20 for DBAbandit) for each workload. The number of trajectories is defined to ensure each victim's training converges. Furthermore, we follow the settings in [19] and implement two variants: (1) **b**: in training and retraining, parameters of the best trajectory for each workload are kept; in inference, the best trajectory is delivered as the recommended index configurations. (2) **m**: in training and retraining, the average parameters of the last 100 trajectories (10 trajectories for DBAbandit) for each workload are kept; in inference, their average performance is reported. We use "IA+implementation" to denote a variant. For example, "DQN-b" means index configuration is based on DQN's best trajectory. The default SWIRL is trained and retrained using **b**, and in inference, SWIRL can directly recommend indexes for different workloads without producing any trajectory. This gives us seven victims in total.

Workloads. Unless otherwise stated, to train and evaluate victim IAs, we follow [19] to generate the normal workloads. Specifically, in each run, we create a workload of N queries, where $N = 18$ in TPC-H and $N = 90$ in TPC-DS, by populating all available query templates of the benchmark and randomly specifying the query frequencies according to a uniform distribution. The number of probing epochs $P = 20$, and the size of a probing or an attacking workload is the same as the normal workloads, i.e., $N_p = N_a = 18$ in TPC-H and $N_p = N_a = 90$ in TPC-DS. The index size budget $B = 4$, i.e., the maximal number of indexes for an IA. The number of specified columns $|\{c\}| = 4$ for IABART.

Evaluation Metrics. We define RR (Reward Reduction) as the evaluation metric to measure the performance of poisoning attacks,

$$RR = \frac{1}{M} * \sum_{i=1}^{M} \frac{c(\mathbb{W}^i, d, \tilde{\mathcal{IA}}(\mathbb{W}^i, d)) - c(\mathbb{W}^i, d, \mathcal{IA}(\mathbb{W}^i, d))}{c(\mathbb{W}^i, d, \emptyset) - c(\mathbb{W}^i, d, \mathcal{IA}(\mathbb{W}^i, d))}, \quad (7)$$

where RR is averaged over M runs. In each run i, an attacking workload is injected by the attacker. $\mathcal{IA}$ is the victim IA before the attack, $\tilde{IA}$ is the victim IA retrained on the normal workload $\mathbb{W}^i$ and the attacking workload. $c(\mathbb{W}^i, d, \mathcal{IA}(\mathbb{W}^i, d))$ is the cost of executing the normal workload on the dataset using the IA's recommended index configuration. RR measures how much the attack makes the victim IA lose its effectiveness in improving performance. $RR \in (-\infty, 1]$. According to Definition 2.2, a negative RR suggests the attack is invalid, i.e., the indexing performance after the attack is higher. A larger RR indicates a more effective attack.

[4]http://www.tpc.org/tpch/
[5]http://www.tpc.org/tpcds/
[6]A trajectory is a path of index selections the IA agent produces by interacting with the environment.

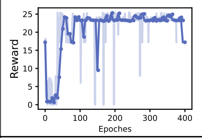
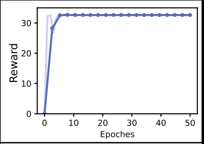
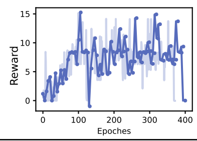
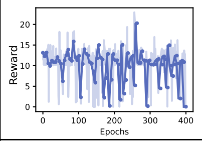
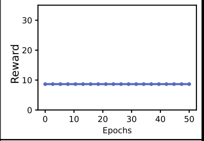
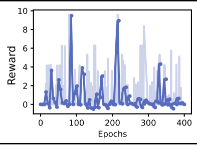
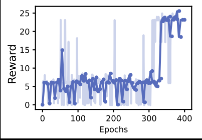
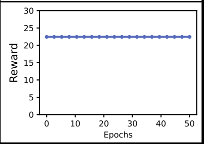
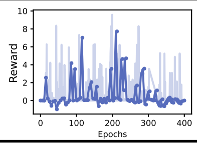
🟥 *optimal* 🟦 *mid-ranked* 🟩 *low-ranked*	***(a) DQN***		***(b) DBAbandit***		***(c) DRLindex***		***(d) SWIRL***	
	Recommended Index	**Trajectory Rewards**	**Recommended Index**	**Trajectory Rewards**	**Recommended Index**	**Trajectory Rewards**	**Directly Predict**	**Retrain before Predict**
Before Poisoning	l_orderkey l_partkey l_suppkey ps_suppkey		l_shipdate l_partkey l_suppkey l_shipmode		l_orderkey l_partkey l_suppkey ps_suppkey		l_partkey l_suppkey ps_suppkey o_custkey Reward: 20.48	l_partkey l_suppkey ps_suppkey o_custkey Reward: 20.48
Mid-ranked Indexes Poisoning	o_custkey l_suppkey l_orderkey n_name		o_custkey o_orderkey o_orderdate p_partkey		l_extendprice o_custkey ps_suppkey r_name		l_suppkey l_orderkey ps_suppkey c_nationkey Reward: 9.34	l_suppkey l_orderkey l_partkey c_nationkey Reward: 17.64
Low-ranked Indexes Poisoning	l_orderkey l_partkey l_suppkey ps_suppkey		l_orderkey l_partkey l_shipmode o_custkey		l_partkey l_suppkey p_retailprice r_name		l_partkey o_custkey c_nationkey c_custkey Reward: 15.89	l_partkey o_custkey l_suppkey c_custkey Reward: 18.58

Figure 8: Cases of attacking DQN, DRLindex, DBAbandit, SWIRL on TPCH-10GB.

Implementation. The database server is a workstation with two Intel Xeon Platinum 8375C 2.90GHz CPUs and PostgreSQL 12.5[7]. The machine learning models, including the victim IAs and IABART, are implemented in a GPU server with Intel Xeon Gold 6133 @ 2.50GHz CPU and eight GeForce RTX 3090 Ti graphics cards. Our codes are available online [8].

6.2 Main Result

Baselines. To verify the attack performance, we compare PIPA to two *competitors.* (1) FSM: each query in the attacking workload is generated by Finite State Machine [43] with a random seed, where each query is assigned a unit frequency. (2) TP: each query is generated by populating the query templates with a random seed, and each query is assigned a frequency that is drawn from the uniform distribution as in [19]. We also compare to *variants* of PIPA. They are different in the usage of the probing information. (3) I-R: Each query in the attacking workload is generated by IABART with randomly specified columns. (4) I-L: We used IABART to generate attacking queries using low-ranked columns, i.e., the bottom 50% columns in the estimated action preference.

Experimental Procedure. We performed poisoning attacks against the seven victim models using different attack methods on the *TPC-H* and *TPC-DS* benchmarks with 1GB and 10GB data. Each attack is repeated for $M = 10$ runs. The boundary of the mid-ranked index interval is [5,1/4L], where L is the length of columns among the dataset. We will explain the choice of interval boundaries in Section 6.4. The results are shown in Figure 7.

Stability of PIPA's attack performance. (1) Our experiments show that *only* PIPA *always achieves positive RR on all datasets against various victim IAs.* Other competitors and model variants can not sustain positive RR values. For example, TP and I-R boost SWIRL's performance by up to 40% on TPC-H 1GB after the attack (i.e., invalid attack). This phenomenon indicates that a robust attack can not be achieved without proper probing and attacking strategies. In particular, PIPA significantly outperforms I-R under almost all scenarios, which is based on the same IABART, emphasizing the need for probing information about the victim. (2) PIPA consistently achieves the highest mean RR averaged over runs (e.g., the orange line in Figure 7). The mean RR averaged over runs achieved by PIPA is $0.2 - 0.6$ higher than FSM and TP, $0.05 - 0.4$ higher than I-R and I-L depending on datasets and victim IAs. (3) In most cases, the performance of PIPA has the least variance (e.g., narrower box in Figure 7), indicating that PIPA is robustly better than the competitors and model variants.

The effect of IABART. The model variants, i.e., PIPA, I-R, I-L, generally outperforms the competitors. This is because IABART can generate queries that satisfy certain performance constraints on the specified index columns. This means that the queries generated by IABART are more purposely designed to attack and have more impacts on the parameters of the IA. This phenomenon suggests that an effective attack can not be achieved without a query generator specifically designed to meet index-aware constraints.

The effect of targeting mid-ranked indexes. By comparing PIPA (targeting mid-ranked indexes) with I-L (targeting low-ranked indexes), we find that the average RR of PIPA significantly improves over I-L in most cases. Further analysis reveals 2 reasons.

Targeting mid-ranked indexes is more effective if the victim employs heuristic index candidate filtering. Heuristic index candidate filtering can help an IA quickly filter out indexes that are not appropriate to the current workload. If targeting low-ranked indexes (i.e., I-L), the heuristic index candidate filtering mechanism can potentially eliminate the bad indexes that I-L attempt to uplift, and thus the attacking effect of I-L is diminished. For example, we find that three low-ranked indexes are selected by I-L, i.e., "c_phone,o_-retailprice,c_custkey", when attacking DQN and SWIRL, only "c_custkey" can successfully achieve the attack effect because

[7]https://www.postgresql.org/
[8]https://anonymous.4open.science/r/PIDPA-73DB

"c_phone" and "o_retailprice" are removed by DQN's heuristic index candidate selection and SWIRL's invalid action masking mechanism.

Targeting mid-ranked columns traps a victim in the local optimum. We corroborate our analysis with the following two cases. The first case is shown in Fig 8(a). After attacking by PIPA, the reward of the victim DQN stays around 10, which is a local optimum. After attacking by I-L, on the other hand, the reward is around 0-5, which is unsatisfying, so DQN tries to leave by random exploration and then jumps out of the bad solution. Eventually, it successfully arrives at the optimal index "l_partkey" after 320 learning epochs. The second case is shown in Fig 8(b). After attacking by PIPA, the reward of the available index arms "o_custkey","p_partkey" stays around 9, which is a local optimum and thus does not trigger the index arm update operation of DBAbandit, so that eventually DBAbandit picks the local optimum solution. However, after attacking by I-L, the index arms are too bad (zero rewards), thus triggering DBAbandit to update the index arms, and finally, DBAbandit correctly selects the arm containing the global optimal index "l_partkey".

Comparison across index advisors. *DRLindex is generally the most vulnerable to poisoning attacks.* For example, three attackers achieved $RR = 0.96$ against DRLindex-b on TPC-DS 1GB. One possible reason for DRLindex's vulnerability is the sparse state representation. The state matrix in DRLindex indicates which column is operated in which query. For example, the matrix is 90×432 in size on TPC-DS. The model will tend to ignore the zero entries (e.g., missing columns in the normal workload) in the sparse matrix. Thus, if the attacking workload operates on a different set of columns from the normal workload, the model parameters will dramatically change, and DRLindex's performance will be severely damaged. Another possible reason is the over-sensitive reward function. DRLindex uses $1/c(\mathbb{W}, d, \mathbb{I})$ as a reward. A small difference in the execution cost c will cause the loss function to vibrate. Thus, the attack can take effect if the attacking workload increases the execution cost.

Poisoning attacks can be better defended by running trial trajectories. We corroborate our analysis in two aspects. (1) DQN, DBAbandit, and DRLindex need multiple trial trajectories to give the final index selection result. Nonetheless, they can be treated as one-off IAs if the first trajectory is adopted as the index advice. As shown in Figure 8(a)-(c), the highest reward is not obtained by the initial trial at epoch=1. With more trial epochs, a larger award can be obtained, and the attack effect is less significant. (2) We also demonstrate the importance of trial trajectories using SWIRL. Note that SWIRL is a one-off model and does not produce multiple trial trajectories for a given workload. Thus, for our demonstration purpose, we re-re-train SWIRL using the normal workload after poisoning (i.e., SWIRL has gone through three training stages). As shown in Fig 8 (d), if SWIRL is to directly predict indexes, both PIPA and I-L cause SWIRL to miss an optimal index "l_suppkey"/"l_partkey". However, when SWIRL is re-re-trained, it selects the optimal indexes (with the reward back to 17.6). We want to highlight that even in this case against a strong victim with extensive training cost, PIPA is effective and outperforms I-L.

The impact of dataset. On TPC-DS dataset, model variants I-R and I-L sometimes achieve comparable attacking performance with PIPA. The underlying is that TPC-DS contains more candidate columns, and some of the victim IAs are less effective in index selection. Thus, once a critical column index is knocked out, the attack will be effective, and the RR will be the same.

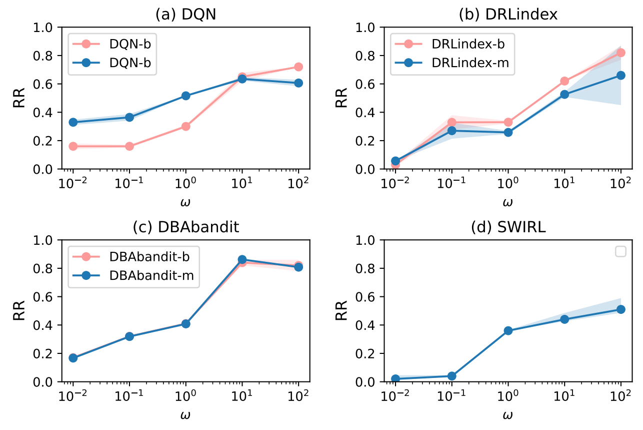


Figure 9: Attacking performance w.r.t. different ω

6.3 Attacking Workload Size

Naturally, the size of the attacking workload N_a has a significant impact on the attacking performance. To conveniently compare the performance among different training workloads, we fix the number of queries in the attacking workload to be $N_a = 180$ and change the number of queries in the normal workload. We compute $\omega = N_a/|\mathbb{W}|$ to measure the poisoning proportion. A small value of ω corresponds to the attacking scenario where the victim IA aims to optimize for all users, i.e., the victim IA will sample a small portion of each user's workload to construct the training set. In this case, it is infeasible to inject a large volume of attacking workload to dominate the training set. On the contrary, a large value of ω means that the victim IA can consider all the injected workloads, e.g., an IA that aims to optimize the overall workload efficiency.

We vary $\omega = 0.01, 0.1, 1, 10, 100$ and repeat the experiments five times. As shown in Figure 9, (1) the attacking performance increases significantly with ω increase. (2) PIPA is valid (i.e., $RR > 0$) even under the smallest ω against all victims. (3) When ω is small, PIPA is most effective against DQN-m, i.e., $RR = 0.32$ when $\omega = 0.01$. Further analysis reveals that DQN is very prone to overfitting to the local optimum, even when a small attacking workload pollutes it. (4) When ω is high, SWIRL has shown strong resistance to poisoning attacks due to its invalid action masking mechanism, which masks off irrelevant columns that are not involved in the training workloads.

6.4 Boundaries of the Target Segment

We investigate the impact of boundary parameters by attacking DQN on TPC-H 10GB. In Section 5, the top-ranked indexes are defined as the best index and its foreign keys. To verify this strategy, we first fix the length of the mid-ranked index interval to 4, i.e., $l_{q-3}, l_{q-2}, l_{q-1}, l_q$, and vary the start point $q - 3 = 2, 3, 4, 5, 6, 7$. We conducted five replicate experiments, and the experimental results and variance are shown in Figure 10(a).

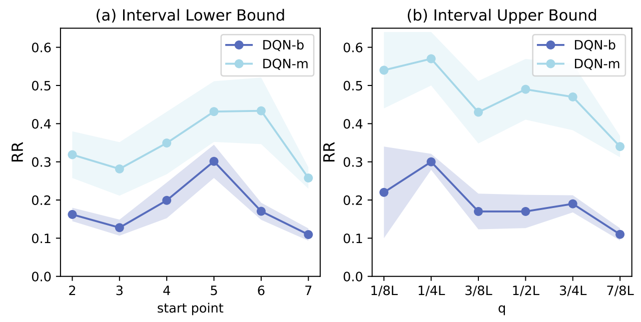


Figure 10: Impact of the mid-ranked index interval

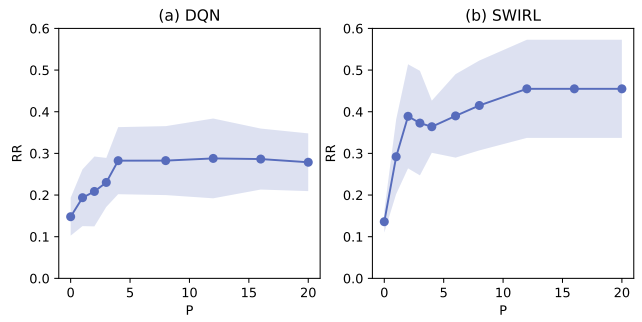


Figure 11: Impact of the number of probing epochs

We find that the best attack effect is achieved when the start point is 5. Upon further analysis, we discovered that when the start point is 5, the top-ranked indexes include "l_partkey" (Rank 1) and its foreign keys "ps_partkey" (Rank 2) and "p_partkey" (Rank 4). This suggests that to optimize the attacking effect, the target segment must exclude the best index and its related foreign keys.

Then we vary $q = 1/8L, 1/4L, 3/8L, 1/2L, 3/4L, 7/8L$, where $L = 61$ is the number of indexable columns in TPC-H 10GB dataset. The experiments are repeated for five runs. As shown in Figure 10(b), we find that the best attack effect is achieved at $q = 1/4L$, yielding the highest mean RR and the smallest variance. When $q = 1/8L$, the variance is significantly larger because the interval of mid-ranked indexes is too small, and the attack effect is uncertain. This observation verifies our assumption in sampling columns from a target segment instead of fixing a set of columns to introduce diversity. When $q > 1/4L$, the attack performance decreases as q increases. When $q = 7/8L$, the mean RR is close to 0.1 because low-ranked indexes are included, and they are ineffective for attacking. The above observation implies that the target segment should be placed on mid-ranked indexes.

6.5 Probing Epochs

To investigate the impact of the probing budget, i.e., the number of probing epochs, we conducted experiments by changing probing epochs $P = 0 - 20$ on TPC-H 10GB, and the rest of the hyper-parameters unchanged. For each setting, we conducted five replicate experiments.

We compare the attack performance on two different types of index advisors (IAs): *one-off* IAs that directly predict the optimal indexes for a given workload [19], and *trial-based* IAs that implement trial runs with different index configurations to find the best ones [20, 29, 30]. Since this distinction may affect the number of probing epochs needed for the attack, we conduct experiments on both DQN and SWIRL as representatives of each type. Figure 10 shows the experimental results and their variance. The main findings are: (1) The attack performance improves as the number of probing epochs increases because more probing epochs lead to a more accurate estimation of the action preference **k**. (2) However, only a few probing epochs are enough to achieve a significant improvement over random attacks, i.e., $P = 4$ achieves the best attack performance against DQN and $P = 2$ achieves a satisfying *RR* against SWIRL.

6.6 Evaluation of IABART

Evaluation Metrics. IABART is trained with 4, 326 queries constructed as in Section 3.1. To evaluate the performance of index-aware query generation, we randomly select three indexes and a reward threshold and generate a testing query that is absent in the training set. We generate $N = 1000$ testing queries and adopt four measurements.

GAC (Grammar Accuracy) measures whether IABART can generate correct SQL queries that conform to the SQL grammar. $GAC = N(q^c)/N$, where $N = 1000$ is the total number of testing queries, and $N(q^c)$ is the number of correct queries that are executable.

IAC (Index Accuracy) measures whether IABART can generate SQL queries that can be optimized by a specified index set.

$$IAC = \frac{\sum_{q^c} |\mathbb{I}^{q^c} \cap \bar{\mathbb{I}}^{q^c}|}{N(q^c)}, \tag{8}$$

where q^c is a correct query, $\mathbb{I}^{q^c}$ is the specified index input to IABART to generate q^c, $\bar{\mathbb{I}}^{q^c}$ is the indexes selected by SWIRL for q^c. $IAC \in [0, 1]$, a larger IAC suggests IABART can generate queries that are optimized by certain indexes.

RMSE (Root Mean Squared Error) measures whether IABART can generate SQL queries that can achieve a given indexing performance on the specified indexes.

$$RMSE = \sqrt{\frac{1}{N(q^c)} \sum_{q^c} (\mathcal{R}(q^c) - \hat{\mathcal{R}}(q^c))^2}, \tag{9}$$

where q^c represents a correct query, $\mathcal{R}(q^c)$ is the random specified reward threshold for IABART to generate q^c, $\hat{\mathcal{R}}(q^c)$ represents the estimated reward of q^c using SWIRL's recommended index configurations. $RMSE \in [0, \infty)$, a smaller $RMSE$ is better.

Distinct [22] measures the diversity of queries. It computes the ratio of unique tokens in each correct query.

Competitors We compare IABART with four query generation methods. (1) **ST**: We build SQL that contains only WHERE filter clauses and only the specified indexes in the WHERE clauses. For example, given a specified index set C1,C2,C3, we construct the SQL as follows: "SELECT C1,C2 FROM T1 JOIN T2 WHERE T1.C1=a OR T2.C2=b", where C1,C2 are the subset of C1,C2,C3 with maximal joinable columns, and a,b are values randomly sampled from the database. (2) **DT**: for any given index set, first, from the templates pool provided in the benchmark, we select a template whose filter condition contains the most specified indexes. Then we use the template to populate the query. (3) **chatGPT**: We ask chatGPT to generate the SQL query by the prompt as shown in table 1. (4) **IABART w/o PT**: IA-BART without Progressive Training.

Table 1: chatGPT prompt

Role	Content
System	You are a SQL generator that can produce SQL statements for given index constraints. Your answer must be a single SQL statement that can be executed directly without any modifications. This means that: (1) Your answer must not contain any other content besides the SQL statement. No explanations or comments are allowed. (2) Your answer must follow the correct syntax rules for SQL. (3) The value must be a value that does not need to be manually specified, such as a specific numerical value, string, etc.
User	Please try to generate SQL that can be optimized by "XXX" based on the table structure of TPC-H database.

Results. Table 2 demonstrated that (1) IABART achieves GAC= 1, showing that IABART can generate syntactically correct queries. (2) IABART achieves the highest IAC, i.e., 0.085 higher than the best competitor chatGPT. This suggests that IABART can generate queries that meet the index requirement. (3) IABART achieves the lowest RMSE, demonstrating that IABART can accurately capture the relationship among queries, the optimal index sets, and the corresponding performance. (4) IABART achieves the highest Distinct, i.e.,0.009 higher than chatGPT. This suggests that IABART is capable of generating more diverse queries. (5) In addition, by comparing IABART with and without progressive training, we find that the proposed training paradigm is vital in the index-aware query generation problem, i.e., the IAC and RMSE drop significantly without progressive training.

Table 2: Performance of query generation

Method	GAC	IAC	RMSE	Distinct
ST	1.00	0.62	30.41	0.004
DT	1.00	0.24	27.76	0.005
chatGPT	0.82	0.60	33.06	0.033
IABART w/o PT	1.00	0.58	19.26	0.030
IABART	**1.00**	**0.69**	**18.85**	**0.042**

7 RELATED WORK

Index advisors. Most conventional Index Advisors (IA) are based on heuristic algorithms that enumerate possible solutions [6, 8, 31, 40]. Recently, learning-based IAs [19] have been proposed to improve the performance of index selection. Most learning-based IAs [19, 20, 26, 29, 30] follow the Reinforcement Learning (RL) framework, i.e., the IA acts as an agent that interacts with the database (DB) environment. In training, the IA repeatedly produces trial trajectories (i.e., index configurations) on the training workloads to maximize the reward of training trajectories. In each step of the trial trajectory, the IA encodes the state of the current environment (e.g., workload, database, currently chosen indexes) and selects an action (i.e., an index) based on the current state. Different designs of state representations, reward functions, and action space exist. A few learning-based IAs are based on Monte Carlo Tree Search (MCTS) [5, 45], which selects the best indexes by expanding the search tree based on a random sampling of the search space. In addition, some IAs replace specific components of the heuristic algorithms by machine learning techniques, e.g., cost estimator [13], query plan [11], and workload representation [35]. We do not attack IAs based on MCTS or heuristic-based IAs with learning components. The reasons are that the MCTS methods need to calculate for "rollout" nodes of the search tree for each training workload and the learning components of heuristic-based IAs are trained separately and are not involved in index mapping.

Poisoning attacks. Attacking machine learning models can be modifications on the testing samples without modifying the model (i.e., evasion attack [2]) or contaminating the training dataset and retraining the model (i.e., poisoning attack [23]). Regarding the types of the victim model, poisoning attacks can be conducted against RL [1, 23, 24, 27, 28, 37, 44], supervised learning [3, 4, 15], and unsupervised learning models [42]. Existing poisoning attacks against RL can be classified into three groups, (1) manipulate the agent's observation and its reward [1, 24, 28, 37, 44], (2) alter the underlying environment [27], or (3) change the agent's action[23]. PIPA differs significantly from existing attacks as PIPA does not require knowledge of the victim and does not interfere with the agent's reward, environment, or action. Furthermore, PIPA differs from the existing poisoning attack against learned index structures [17], which also assumes a clear-box setting.

SQL query generation. Generally, query generation can be divided into (1) *Random methods* that generate queries by following pre-defined heuristic rules [32, 36], (2) *Template-based methods* that rely on some given SQL templates and tweak the predicate values [7] or change values [25] based on a space-pruning technique to reduce the search space, (3) *Learning-based methods* that employ a reinforcement learning framework [43] to meet specific cardinality or cost constraints. The former two types of methods are not capable of index perception. Thus, although they can generate valid queries, only a small portion of the queries can be used for probing and attacking. It is worth pointing out that the reinforcement learning framework [43] can not be applied with trivial modifications such as specifying the constraint on columns because a separate RL model on each column combination is needed due to the reward determination and the training cost will be too expensive.

8 CONCLUSION

In this paper, we made the first attempt at poisoning attacks against learning-based IAs. First, we proposed to probe the index preference under the opaque-box setting. Second, we designed attacking workloads to trap IAs within local optimum. Besides, we proposed a query generation method for probing and attacking using large language models (BART). Extensive experiments on different benchmarks against four typical learning-based IAs demonstrated the effectiveness of PIPA.

REFERENCES

[1] Vahid Behzadan and Arslan Munir. 2017. Vulnerability of deep reinforcement learning to policy induction attacks. In *Machine Learning and Data Mining in Pattern Recognition: 13th International Conference, MLDM 2017, New York, NY, USA, July 15-20, 2017, Proceedings 13*. Springer, 262–275.

[2] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13*. Springer, 387–402.

[3] Battista Biggio, Blaine Nelson, and Pavel Laskov. 2012. Poisoning attacks against support vector machines. *arXiv preprint arXiv:1206.6389* (2012).

[4] Battista Biggio, Ignazio Pillai, Samuel Rota Bulò, Davide Ariu, Marcello Pelillo, and Fabio Roli. 2013. Is data clustering in adversarial settings secure?. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*. 87–98.

[5] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4, 1 (2012), 1–43.

[6] Nicolas Bruno and Surajit Chaudhuri. 2005. Automatic physical database tuning: A relaxation-based approach. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. 227–238.

[7] Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. 2006. Generating queries with cardinality constraints for dbms testing. *IEEE Transactions on Knowledge and Data Engineering* 18, 12 (2006), 1721–1725.

[8] Surajit Chaudhuri and Vivek R Narasayya. 1997. An efficient, cost-driven index selection tool for Microsoft SQL server. In *VLDB*, Vol. 97. San Francisco, 146–155.

[9] Antonio Emanuele Cinà, Kathrin Grosse, Ambra Demontis, Sebastiano Vascon, Werner Zellinger, Bernhard A Moser, Alina Oprea, Battista Biggio, Marcello Pelillo, and Fabio Roli. 2022. Wild patterns reloaded: A survey of machine learning security against training data poisoning. *arXiv preprint arXiv:2205.01992* (2022).

[10] Tran Khanh Dang, Phat T Tran Truong, and Pi To Tran. 2020. Data poisoning attack on deep neural network and some defense methods. In *2020 International Conference on Advanced Computing and Applications (ACOMP)*. IEEE, 15–22.

[11] Bailu Ding, Sudipto Das, Ryan Marcus, Wentao Wu, Surajit Chaudhuri, and Vivek R Narasayya. 2019. Ai meets ai: Leveraging query executions to improve index recommendations. In *Proceedings of the 2019 International Conference on Management of Data*. 1241–1258.

[12] Minghong Fang, Neil Zhenqiang Gong, and Jia Liu. 2020. Influence function based data poisoning attacks to top-n recommender systems. In *Proceedings of The Web Conference 2020*. 3019–3025.

[13] Jianling Gao, Nan Zhao, Ning Wang, and Shuang Hao. 2022. SmartIndex: An Index Advisor with Learned Cost Estimator. In *Proceedings of the 31st ACM International Conference on Information & Knowledge Management*. 4853–4856.

[14] Hai Huang, Jiaming Mu, Neil Zhenqiang Gong, Qi Li, Bin Liu, and Mingwei Xu. 2021. Data poisoning attacks to deep learning based recommender systems. *arXiv preprint arXiv:2101.02644* (2021).

[15] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li. 2018. Manipulating machine learning: Poisoning attacks and countermeasures for regression learning. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 19–35.

[16] Mandar Joshi, Danqi Chen, Yinhan Liu, Daniel S Weld, Luke Zettlemoyer, and Omer Levy. 2020. Spanbert: Improving pre-training by representing and predicting spans. *Transactions of the association for computational linguistics* 8 (2020), 64–77.

[17] Evgenios M Kornaropoulos, Silei Ren, and Roberto Tamassia. 2020. The Price of Tailoring the Index to Your Data: Poisoning Attacks on Learned Index Structures. *arXiv preprint arXiv:2008.00297* (2020).

[18] Jan Kossmann, Stefan Halfpap, Marcel Jankrift, and Rainer Schlosser. 2020. Magic mirror in my hand, which is the best in the land? an experimental evaluation of index selection algorithms. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2382–2395.

[19] Jan Kossmann, Alexander Kastius, and Rainer Schlosser. 2022. SWIRL: Selection of Workload-aware Indexes using Reinforcement Learning.. In *EDBT*. 2–155.

[20] Hai Lan, Zhifeng Bao, and Yuwei Peng. 2020. An index advisor using deep reinforcement learning. In *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*. 2105–2108.

[21] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. 2019. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461* (2019).

[22] Jiwei Li, Michel Galley, Chris Brockett, Jianfeng Gao, and Bill Dolan. 2015. A diversity-promoting objective function for neural conversation models. *arXiv preprint arXiv:1510.03055* (2015).

[23] Guanlin Liu and Lifeng Lai. 2021. Provably efficient black-box action poisoning attacks against reinforcement learning. *Advances in Neural Information Processing Systems* 34 (2021), 12400–12410.

[24] Yuzhe Ma, Xuezhou Zhang, Wen Sun, and Jerry Zhu. 2019. Policy poisoning in batch reinforcement learning and control. *Advances in Neural Information Processing Systems* 32 (2019).

[25] Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. 2008. Generating targeted queries for database testing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 499–510.

[26] R Malinga Perera, Bastian Oetomo, Benjamin IP Rubinstein, and Renata Borovica-Gajic. 2021. DBA bandits: Self-driving index tuning under ad-hoc, analytical workloads with safety guarantees. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE, 600–611.

[27] Amin Rakhsha, Goran Radanovic, Rati Devidze, Xiaojin Zhu, and Adish Singla. 2020. Policy teaching via environment poisoning: Training-time adversarial attacks against reinforcement learning. In *International Conference on Machine Learning*. PMLR, 7974–7984.

[28] Amin Rakhsha, Xuezhou Zhang, Xiaojin Zhu, and Adish Singla. 2021. Reward poisoning in reinforcement learning: Attacks against unknown learners in unknown environments. *arXiv preprint arXiv:2102.08492* (2021).

[29] Zahra Sadri, Le Gruenwald, and Eleazar Lead. 2020. DRLindex: deep reinforcement learning index advisor for a cluster database. In *Proceedings of the 24th Symposium on International Database Engineering & Applications*. 1–8.

[30] Zahra Sadri, Le Gruenwald, and Eleazar Leal. 2020. Online index selection using deep reinforcement learning for a cluster database. In *2020 IEEE 36th International Conference on Data Engineering Workshops (ICDEW)*. IEEE, 158–161.

[31] Rainer Schlosser, Jan Kossmann, and Martin Boissier. 2019. Efficient scalable multi-attribute index selection using recursive strategies. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1238–1249.

[32] Andreas Seltenreich, Bo Tang, and Sjoerd Mullender. 2020. SQLsmith: Score list.

[33] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P Adams, and Nando De Freitas. 2015. Taking the human out of the loop: A review of Bayesian optimization. *Proc. IEEE* 104, 1 (2015), 148–175.

[34] Ankur Sharma, Felix Martin Schuhknecht, and Jens Dittrich. 2018. The case for automatic database administration using deep reinforcement learning. *arXiv preprint arXiv:1801.05643* (2018).

[35] Tarique Siddiqui, Saehan Jo, Wentao Wu, Chi Wang, Vivek Narasayya, and Surajit Chaudhuri. 2022. ISUM: Efficiently Compressing Large and Complex Workloads for Scalable Index Tuning. In *Proceedings of the 2022 International Conference on Management of Data*. 660–673.

[36] Donald R Slutz. 1998. Massive stochastic testing of SQL. In *VLDB*, Vol. 98. Citeseer, 618–622.

[37] Yanchao Sun, Da Huo, and Furong Huang. 2020. Vulnerability-aware poisoning mechanism for online rl with unknown dynamics. *arXiv preprint arXiv:2009.00774* (2020).

[38] Loc Truong, Chace Jones, Brian Hutchinson, Andrew August, Brenda Praggastis, Robert Jasper, Nicole Nichols, and Aaron Tuor. 2020. Systematic evaluation of backdoor data poisoning attacks on image classifiers. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*. 788–789.

[39] Eric Wallace, Tony Z Zhao, Shi Feng, and Sameer Singh. 2020. Concealed data poisoning attacks on nlp models. *arXiv preprint arXiv:2010.12563* (2020).

[40] Kyu-Young Whang. 1987. Index selection in relational databases. *Foundations of Data Organization* (1987), 487–500.

[41] Wentao Wu, Chi Wang, Tarique Siddiqui, Junxiong Wang, Vivek Narasayya, Surajit Chaudhuri, and Philip A Bernstein. 2022. Budget-aware Index Tuning with Reinforcement Learning. In *Proceedings of the 2022 International Conference on Management of Data*. 1528–1541.

[42] Chaofei Yang, Qing Wu, Hai Li, and Yiran Chen. 2017. Generative poisoning attack method against neural networks. *arXiv preprint arXiv:1703.01340* (2017).

[43] Lixi Zhang, Chengliang Chai, Xuanhe Zhou, and Guoliang Li. 2022. Learnedsqlgen: Constraint-aware sql generation using reinforcement learning. In *Proceedings of the 2022 International Conference on Management of Data*. 945–958.

[44] Xuezhou Zhang, Yuzhe Ma, Adish Singla, and Xiaojin Zhu. 2020. Adaptive reward-poisoning attacks against reinforcement learning. In *International Conference on Machine Learning*. PMLR, 11225–11234.

[45] Xuanhe Zhou, Luyang Liu, Wenbo Li, Lianyuan Jin, Shifu Li, Tianqing Wang, and Jianhua Feng. 2022. Autoindex: An incremental index management system for dynamic workloads. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 2196–2208.