Robustness of Updatable Learning-based Index Advisors against Poisoning Attack

Yihang Zheng  
Institute of Artificial Intelligence, Xiamen University  
Xiamen, China  
yihang@stu.xmu.edu.cn

Chen Lin∗  
School of Informatics, Xiamen University & Shanghai Artificial Intelligence Laboratory  
Xiamen, China  
chenlin@xmu.edu.cn

Xian Lyu  
School of Informatics, Xiamen University  
Xiamen, China  
xianlyu2023@stu.xmu.edu.cn

Xuanhe Zhou  
Department of Computer Science, Tsinghua University  
Beijing, China  
zhouxuan19@mails.tsinghua.edu.cn

Guoliang Li  
Department of Computer Science, Tsinghua University  
Beijing, China  
liguoliang@tsinghua.edu.cn

Tianqing Wang  
Huawei Company  
Beijing, China  
wangtianqing2@huawei.com

∗Chen Lin is the corresponding author.

ABSTRACT

Despite the promising performance of recent learning-based Index Advisors (IAs), they exhibit the robustness issue when training data was polluted by poisoning attacks. This paper presents the first attempt to study the robustness of updatable learning-based IAs against poisoning attack, i.e., whether the IAs can maintain robust performance if their training/updating is disturbed by injecting an extraneous toxic workload. The goal is to provide an opaque-box stress test that is generally effective in evaluating the robustness of different learning-based IAs without using the users’ private data. There are three challenges, i.e., how to probe “index preference” from opaque-box IAs, how to design effective injecting strategies even if the IAs can be fine-tuned, and how to generate queries to meet with the specific constraints for IA probing and injecting. The presented stress-test framework PIPA consists of a probing stage, an injecting stage, and a query generator. To address the first challenge, the probing stage estimates the IA’s indexing preference by observing its responses to the probing workload. To address the second challenge, the injecting stage injects workloads that spoof the IA to demote the top-ranked indexes in the estimated indexing preference and promote mid-ranked indexes. The stress-test is effective because the IA 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, and that existing learning-based IAs are non-robust when faced with even a subtle amount of injected extraneous toxic workloads.

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, 39] have been proposed to guide the search of possible indexes. Since the heuristics algorithms are manually designed and are rarely altered, their capabilities are usually limited, i.e., they may miss some beneficial indexes that are suitable for a particular workload/data. On the contrary, learning-based Index Advisors (IAs) [19, 20, 26, 29, 30, 33, 40, 45] can update their indexing strategies by optimizing the model parameters with training data. This approach has recently gained significant attention from both academia and industry.

For learning-based IAs, the quality of training data determines the ability of machine learning methods. However, in realistic scenarios, there is usually a non-negligible and inevitable gap between the training data and the testing data. If the training data is updated, the learning-based IA models will be updated which will affect the performance on the testing data. Thus there is a risk that if there is malicious poisoning attack on the training workload, whether the IA models are robust? (i.e., the performance on the testing workload will be significant degrade.) Suppose a supplier with multiple franchisees (e.g. tenants of Salesforce) sharing the same cloud business solution. Learning-based IAs are applied to normal daily workloads submitted by franchisees and employees. Suppose an attacker (e.g., a reckless employee or a malicious franchisee) submits abnormal workloads at some point before the IAs update the parameters. In that case, the training data of IAs contains extraneous toxic workloads. Figure 1 shows that the extraneous workloads in training data can be toxic (e.g., in red) and significantly damage the IAs’ performance. With only 1% extraneous toxic workloads, the execution cost of the same testing workloads by IAs’ indexes is increased by 20%. Thus poisoning attack on IAs can be utilized by malicious
attacker (e.g., fired employees and hostile competitors). It’s of significant importance to study the robustness of learning-based IAs against poisoning attacks.\(^1\)

The above illustration raises concerns about the robustness of updatable learning-based IAs when training data is polluted, which refers to the capability of maintaining the performance under the same testing workload when the training/updating is disturbed by extraneous toxic workloads. This investigation yields two main advantages. First, it sheds insight into the training and updating of learning-based IAs (e.g., how to choose appropriate training/updating workloads and how to design appropriate training/updating strategy). Second, it facilitates the database administrators (DBAs) to deploy a more robust leaning-based IA so as to ensure that the indexing performance is reliable and hardly affected by model updates with diversified training workloads.

To assess the aforementioned robustness, the core idea is to inject extraneous toxic workloads into the training workloads and compare the performance before and after the injection, which is widely studied and referred to as “poisoning attack” in Artificial Intelligence (AI) fields [9, 10, 12, 14, 37, 38]. As illustrated in Figure 1, the IA (DQN) is first trained and tested on the normal workloads (green flow). After it is re-trained on the set of normal and injected workloads (red flow), it recommends inaccurate indexes on the normal workloads, reflecting the lack of robustness of DQN.

Despite the similar workflow, the robustness test of updatable learning-based IAs is distinctive in two aspects that existing poisoning attack methods are not applicable. First, a poisoning attack injects incorrect training data (e.g., with wrong labels or wrong rewards). In contrast, the injected extraneous workloads are executable and sargable (i.e., these workloads can take advantage of indexes) queries to reflect real scenarios. Second, in order to propose a universally effective assessment that is independent of specific learning-based IA models, testing workloads, and training process, we should conduct an opaque-box testing. That is, the evaluator has no access to the IA’s internal designs (e.g., the applied algorithms, the considered index candidates, and the exact values of model parameters). Besides, the evaluator has no knowledge of the IA’s training data (e.g., the normal workloads) and cannot control the training process (e.g., it cannot provide wrong rewards or directly interfere with the index selection).

Under the above settings, the evaluator can only interact with the IA by submitting an input workload and observing the IA’s output. Thus most existing poisoning attack methods are hard to follow, which require a certain amount of knowledge of the training process, such as the gradient [27], rewards [1, 24, 28, 36, 44] action space [23], and the labels of the training data [17]. Specifically, the main challenges are as follows.

C1: IA Probing Strategy. There is an unlimited number of injection workloads to choose from, and we are interested in injections that can cause a significant performance reduction to stress-test the IAs. Obviously, injections without any guidance information cannot effectively function as stress tests. The problem is, what information can be useful in an opaque-box setting? Inputting some probing workloads and extracting information from the output may be a solution. However, due to the cost of executing the probing workloads, it remains a difficult problem to design effective probing workloads to meet the small probing overhead budget.

C2: Effective Injection Strategy. The impacts of injection workloads differ on learning-based IAs with diverse design details. Some IAs, such as SWRL [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. For one-off IAs, the injection workloads explicitly alter their parameters. Others [20, 26, 29, 30] are trial-based for the inference stage, i.e., after being well-trained or updates, the IA still iterates several times to produce trial indexes for a given workload. For the trial-based IAs, the injection workload only alters initialization of their parameters, and the impact on the final performance is far more complicated. It is challenging to design injection strategies that are both effective on one-off and trial-based IAs.

C3: Query Generation for IA Probing and Injecting. A query generator is needed to automate the stress-test procedure. Existing query generators are not applicable because their goals do not match the probing and injection strategies (i.e., to reveal IA’s information or to be most influential on the IA’s training). For example, as shown in Figure 1(3), the queries generated by SQLsmith [32], which focuses on generating queries that satisfy the syntax constraint, cannot expose that DQN is a non-robust IA.

We study the above challenges and propose PIPA (Probing-Injecting Poisoning Attack). PIPA consists of a probing stage before an injecting stage, both of which are based on a query generator IABART (Index Aware BART). To address C1, the probing stage (Section 4) iteratively refines the probing workloads to estimate the IA’s indexing preference on all indexable columns (i.e., which column is likely to be chosen to build an index) within the probing budget. To address C2, the injecting stage (Section 5) designs workloads that can actively spoof the IA to demote the top-ranked indexes in the estimated indexing preference and promote mid-ranked indexes. This strategy is effective both on “one-off” IAs since degrading top-ranked indexes leads to inferior indexing performance and on “trial-based” IAs because promoting mid-ranked effectively traps the IA in a local optimum. To address C3, IABART targets on generating a SQL query that can be optimized by building an index on some given columns (Section 3). We propose several techniques, including progressive training to increase the awareness of indexing performance in the resulting SQL query and a novel decoding method to ensure the results are syntactically correct queries.

---

1Details are shown in the supplementary file in https://anonymous.4open.science/r/PIPA-73DB/Supplementary_Material.md
In summary, our contributions are four-fold. (1) We make the first attempt to study the robustness of updatable learning-based IAs against poisoning attacks. (2) We propose an opaque-box stress-test framework PIPA that can be used to stress-test the robustness of learning-based IAs regardless of the IA’s internal mechanisms, user data, and the testing workloads. (3) We conduct comprehensive experiments on different benchmarks, IAs, and injection workload sizes to demonstrate that the one-off IAs and trial-based IAs are both not robust when training workloads are polluted. (4) We reveal some interesting insights, such as the strong impact of mid-ranked indexes, the local optimum trap in the learning process of IA, and the influence of design details of various IAs on robustness. We believe that these insights can better assist DBAs in deploying learning-based IAs.

2 ROBUSTNESS OF LEARNING-BASED IAS AGAINST POISONING ATTACK

2.1 Problem Definition

Definition 2.1. (Learning-based Index Advisor) Given a target workload \(W\) on a dataset \(d\), an index advisor \(IA\) is a function parameterized by \(\theta\) that aims to build a set of indexes \(I_{renamed} = IA_0(W, d)\), such that the performance improvement of executing the workload \(W\) with indexes \(I_{renamed}\) against the indexes is maximized, under the budget constraint that the index storage/number size \(B(I_{renamed})\) is not larger than a budget bound \(B\), i.e., \(B(I_{renamed}) \leq B\).

Generally, the IA’s parameters are optimized by minimizing a certain loss function computed on the training workloads. For a given IA, its parameters are dependent on the training workloads. Many IAs [19, 20, 26, 29, 30, 33, 40, 45] assume the training workloads are equivalent to the target workload. We follow this setting to establish a performance baseline, i.e., the intended performance of an IA well-trained on the target workload.

Definition 2.2. (Performance Baseline of a Learning-based IA) Given a target workload \(W\), a dataset \(d\), and a learning-based IA \(IA\) with parameters \(\theta\), its performance baseline is defined as the execution cost \(c\) of the target workload based on the well-trained IA’s recommended index which minimizes the loss function \(L(\theta, W)\).

\[c_b(W, d, IA) = c(W, d, IA_0(W, d)), \theta^* \leftarrow \arg\min_{\theta} L(\theta, W)\]

Since \(IA_0\) is updatable, comparing its performance before and after the model update reflects its robustness. To construct the new training workloads to update \(\theta\), we inject an extraneous workload \(\hat{W}\), “extraneous” means \(\hat{W} \cap W = \emptyset\). We adopt the injection for three reasons. (1) Injection mimics real scenarios, e.g., the training workloads contain historical workloads that are not present in the current target workload. (2) Injection is also adopted in some IAs [19], e.g., the training workloads contain variants of the target workload. (3) We can control the degree of overlap between previous and new training workloads by setting the size of injection workloads. Accordingly, we define the Absolute performance Degradation (AD) as a measure of the learning-based IA’s robustness when its training is disturbed by various extraneous injection workloads.

\[\text{Definition 2.3. (Absolute performance Degradation)}\] Given the IA’s performance baseline \(c_b\), the extraneous injection workload \(\hat{W}\) selected by the evaluator, the IA is retrained on the combined set \(\{W, \hat{W}\}\) to update the parameters, the Absolute performance Degradation (AD) is defined as the relative increased execution cost,

\[AD(W, d, IA, \hat{W}) = \frac{c(W, d, IA_0(W, d)) - c_b(W, d, IA)}{c(W, d, \hat{W}) - c_b(W, d, IA)}\]

For heuristic IAs, the AD score is always zero. As the testing workload \(W\) does not change, the indexing recommendation results generated according to the predefined procedure will remain unchanged. For some extraneous injection workloads, such as the workload variants adopted by SWIRL [19], the AD score is usually negative. We argue that the AD score should be positive for a correct evaluator. If the AD score is negative, it means that the injection workload can not disturb the training process. Thus, this AD score can not truly evaluate the robustness of IAs against poisoning attacks. Instead, high positive AD scores from various injection workloads (especially the maximal score among them) reflect that the IA is non-robust.

Clearly, there is an unlimited number of injection workloads \(\hat{W}\). To develop a thorough evaluation against the extreme condition inspired by poisoning attack [41], we next define the concept of Toxic Injection Workload that has a destructive impact on the IA’s performance.

Definition 2.4. (Toxic Injection Workload) Given the well-trained IA \(IA_{\emptyset}\) as the victim, if an injection workload harms the updated IA’s performance on the target workloads compared with its performance baseline,

\[c(W, d, IA_0(W, d)) > c_b(W, d, IA), \theta^* \leftarrow \arg\min_{\theta} L(\theta, \{W, \hat{W}\})\]

then \(\hat{W}\) is a toxic injection workload.

Toxic injection workloads are a subset of all injections. Compared to random injections, toxic injection workloads can stress-test the robustness limits of IA on a larger scale. Consequently, we define another robustness metric.

Definition 2.5. (Relative performance Degradation) For each IA, various random injection workloads \(\hat{W}\), Relative performance Degradation (RD) calculates the difference between the performance degradation caused by random injection workloads and toxic injection workloads:

\[RD(W, d, IA, \hat{W}) = AD(W, d, IA, \hat{W}) - AD(W, d, IA, \hat{W})\]

where \(AD(W, d, IA, \hat{W})\) acts as performance testing for the IA under different training conditions, \(AD(W, d, IA, \hat{W})\) acts as stress test under deliberately malfunctioning training, and RD measures to what extent the performance degradation under toxic injection workload exceeds the expected degradation. Since one would expect the injection in training data to cause performance degradation, RD complements AD to provide a more comprehensive measurement of the robustness of learning-based IAs. Since there is a vast possibility of injection workloads, in the experiments, we use average AD and average RD over multiple runs.
2.2 PIPA Overview

From Definition 2.5, the measurement of the robustness is through the toxic injection workload, which is generated by PIPA (Probing-Injecting Poisoning Attack). For a more generalized robustness evaluation, we impose certain restrictions on the evaluator’s ability. (1) To generalize to all learning-based IAs, the evaluator cannot observe the internal design details of IA; only the input and output interfaces of IA are exposed. (2) To be independent of users’ private information, the evaluator is only able to access the schema of the database, such as the structure of tables and columns, but cannot obtain the private data.

**Workflow.** PIPA is composed of three modules, i.e., query generator IABART, IA probing, IA injecting, as shown in Figure 2. The training of IABART is independent of the other two modules. In each evaluation, the evaluator implements the probing stage before the actual injecting stage, and IABART is called in both stages.

**Index Advisor Probing.** Since the model parameter \( \theta \) and training workloads \( W \) are unknown under the opaque-box setting, we extract the IA’s indexing preferences, i.e., which column is more likely to be chosen by the IA regardless of the target testing workloads. Intuitively, the preferred columns will likely be effective on the target testing workloads. As shown in Figure 2, in each iteration, the probing workload is renewed to obtain an accurate estimate of indexing preference (details in Section 4.3).

**Index Advisor Injecting.** We can use the indexing preference to supervise the injection. Specifically, as shown in Figure 2, we can generate 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 workloads and re-training the IA will eventually down-weight the preferred columns in \( \theta \). As a result, the updated parameters \( \theta \) will be less effective on target workloads.

**Query Generator.** The probing and injecting 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.

---

**Figure 2: Framework overview of PIPA**

---

**Figure 3: Overview of the training and inference of IABART**

---

### 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 \( \{l_1, \ldots, l_n\} \), the evaluator’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., \( \forall I \) with \( |I| = n, c(q, d, I) \leq c(q, d, l') \).

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 inflexible 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 \( x^{mask} \) corrupted from \( x \) (e.g., some tokens are masked) from both directions (i.e., left to right and right to left) and produces a representation \( E(x^{mask}) \). The auto-regressive decoder outputs a sequence \( y \) that recovers the input text from left to right, based on the encoder’s representation and previously generated tokens, i.e., \( y_t = D(E(x^{mask}), y_{<t}) \). To utilize the common-sense knowledge learned from pre-training in large-scale English language corpus (e.g., a basic understanding of numerical magnitudes), we initialize the model by BART-base\(^3\).

#### 3.1 Construction of Training Data

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

$q$ is generated by feeding a random seed to the Finite State Machine (FSM) [43] on $d$. We generate each $q$ starting from the state “FROM,” which helps the FSM to determine the table of the SQL statement first in order to determine the subsequent legal column candidates in the next steps. We tokenize the query because a SQL query usually contains words that are rare in the open-domain corpus or in training (i.e., Out of Distribution) 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 handle OOD problems.

To associate different SQL queries with their appropriate index configurations, we use SWIRL [19] to recommend a set of indexes $I^q_d$ 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 $R(I^q_d)$ in the training sample is to help IABART to understand the benefit of $I^q_d$ and further enhance its accuracy in generating a SQL query to meet the index requirements. We compute $R(I^q_d) = c(q,d,0) - c(q,d,\emptyset)$, where $\emptyset$ is the null index. We use estimated cost instead of the actual cost to speed up the construction and collect more training samples. $R(I^q_d)$ is discretized into the interval of 0, 1 rounded up to two decimals (e.g., 0.31) so that the numerical performance is treated as a classification task to avoid generating real numbers.

### 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 $x^{\text{mask}}$. Then, the masked span prediction attempts to recover the corrupted sequence by optimizing the following loss function:

$$L(E, D) = \sum_{t=s}^{e} - log p(x_t = D(x^{\text{mask}}), y_{<t}).$$  \hspace{1cm} (4)

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 $I^q_d$. 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 an 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> \_<SEP> <SEP> <MASK>\_", where $\_I$ is 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 sub-tokens in the training samples to handle OOD problems. We propose a prefix-matching strategy to adapt the word-based FSM to our sub-token level 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_person” are reserved for future matching. Next, the decoder updates its output. “_” 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.

### 4 PROBING INDEX PREFERENCE

#### 4.1 Derivation of Indexing Preference

We only extract information regarding single-column indexes in the probing stage for practical reasons. (1) The internal architecture of the 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.

We seek to derive a ranking over all indexable columns as the indexing 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 \( \theta \) is unknown. Second, the ranking order (i.e., which indexable column is preferred over the other) provides substantial supervision in generating the injection workload.

To determine the order position of each column, our key idea is to compute the expectation of the “preference” of each indexable column by aggregating over different probing workloads. The intuition is illustrated in Figure 4. Generally, let’s denote the IA chooses a column \( l_i \) for the target workload \( \mathcal{W} \) based on \( \theta(l_i, \mathcal{W}) \). Figure 4(a) illustrates \( \theta(l, \mathcal{W}) \) on the specific \( \mathcal{W} \). Note that the exact value of \( \theta(l, \mathcal{W}) \) is invisible because we have no access to the user data and the target workloads. Nonetheless, if we have multiple probing workloads \( \mathcal{PW} \) and Figure 4(b)-(c) illustrates \( \theta(l_i, \mathcal{PW}) \) on each slice. Although this is still invisible, we can approximate it by observing the IA’s output index \( \theta(l_i, \mathcal{PW}) \) and incorporating the benefit of taking each index \( \mathcal{R}(l, \mathcal{PW}) \). Based on different probing workloads, we can then calculate the expectation, i.e., \( \mathcal{K}(l_i) = \mathbb{E}_{\mathcal{PW}} \left[ \theta(l_i, \mathcal{PW}) \mathcal{R}(l_i, \mathcal{PW}) \right] \) and eliminate the workload variable, plotted as Figure 4(d). The resulting \( \mathcal{K}(l_i) \) represents a tendency that the IA favors a column regardless of the workloads. Thus, it allows us to be unaware of the training details, e.g., the training and target 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.

The above reason gives the definition of indexing preference \( k \),

\[
k = \langle l_1, \ldots, l_L \rangle,
\]

\[
\forall i < j, \quad \mathcal{K}(l_i) > \mathcal{K}(l_j),
\]

\[
\mathcal{K}(l_i) = \mathbb{E}_{\mathcal{PW}} \left[ \theta(l_i, \mathcal{PW}) \mathcal{R}(l_i, \mathcal{PW}) \right],
\]

where \( k \) is a ranking over the set of indexable columns \( l_1, \ldots, l_L \). \( \mathcal{R}(l, \mathcal{PW}) \) is the benefit of building an index on column \( l \) for a particular probing workload \( \mathcal{PW} \). \( \theta(l_i, \mathcal{PW}) \) is the approximated parameter based on observed indexes, \( \mathbb{E}_{\mathcal{PW}} \) is the expectation over all probing workloads.

### 4.2 Calculation of Indexing Preference

Suppose the probing stage repeats for \( P \) iterations, each iteration generates a probing workload \( \mathcal{PW} \). \( 1 \leq p \leq P \) that contains \( N_p \) probing queries, the IA’s output index configuration is \( \mathcal{IP} \).

First, to approximate parameter \( \theta(l_i, \mathcal{PW}) \), we know that in each inference step \( t \), the IA outputs an index with the maximal \( \theta \). It means that if \( l_i \in \mathcal{IP} \), then \( \forall l_j \notin \mathcal{IP} \), \( \theta(l_i, \mathcal{PW}) > \theta(l_j, \mathcal{PW}) \). Also, since each column appears at most once in the output indexes recommended by a well-train IA, this is equivalent to using a sparse policy \( \theta \),

\[
\theta(l_i, \mathcal{PW}) = \begin{cases} 
1 & l_i \in \mathcal{IP}, \\
0 & l_i \notin \mathcal{IP}.
\end{cases}
\]

Next, we assume each index contributes equally to the indexing benefit. Most IAs [19, 20, 26] adopt the relative cost reduction to quantify the indexing benefit and use it as the loss function.

\[
\mathcal{R}(l_i, \mathcal{PW}) = \begin{cases} 
1 - \frac{c(\mathcal{PW}, d, \mathcal{IP})}{c(\mathcal{PW}, d, \mathcal{IP})}, & \text{if } l_i \in \mathcal{IP} \\
0, & \text{else},
\end{cases}
\]

where \( c(\mathcal{PW}, d, \mathcal{IP}) \) is the actual execution cost of the probing workload using the IA’s recommended index configuration, \( \theta \) is the null index, \( |\mathcal{IP}| \) is the number of indexes in the output.

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

\[
\mathcal{K}(l_i) = \frac{1}{P} \sum_{P, \mathcal{IP}} \frac{1 - \frac{c(\mathcal{PW}, d, \mathcal{IP})}{c(\mathcal{PW}, d, \mathcal{IP})}}{|\mathcal{IP}|}.
\]

### 4.3 Probing Strategy

Clearly, the order of \( k \) is affected by probing workloads \( \mathcal{PW} \). Unfortunately, given the probing budget \( P, N_p \) for the number of probing epochs and the size of the probing workloads, it is neither practical nor necessary to enumerate all possible probing workloads. Our goal is to obtain an accurate ranking within the probing budget. The intuition is illustrated in Figure 5. Suppose there are three columns \( l_1, l_2, l_3 = \text{"orderID"}, \text{"userID"}, \text{"productID"} \) and their \( \mathcal{R}(l, \mathcal{PW}) \) are visualized as three lines in Figure 5, where the x-axis denotes the workloads. We highlight three areas in the x-axis, i.e., workloads that the three indexing columns can optimize. Note that these lines are unknown and if in a probing round, a probing workload \( \mathcal{PW} \) is executed, a point \( \mathcal{R}(l, \mathcal{PW}) \) can then be observed. Suppose, based on the original three observations \( l_1, l_2, l_3 \), the ranking order is “orderID” > “userID” > “productID”. In the next round, to collect new information, we prefer to generate a probing workload that likely changes the previous 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 IA’s output \( \mathcal{IP} \) is likely to include “productID”, and the observation 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., IA, dataset), query generator IABART and hyper-parameters 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 \( \mu \) 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 \( \mathcal{PW} \) that can be optimized by building indexes on \( \{c\} \) (line 5). Let the IA recommend index configuration \( \mathcal{IP} \) for \( \mathcal{PW} \) on dataset \( d \) (line 7). For each column in the index configuration \( \mathcal{IP} \), the relative cost reduction is observed to update \( \mathcal{K} \).
Figure 4: Intuition of probing and injecting

Figure 5: Illustration of the Probing Strategy

Algorithm 1: Probing procedure

Data: An opaque-box $IA$, the dataset $d$, the query generator $IABART$, probing budget $P$, $N_p$, number of specified columns $|\{c\}|$

Result: The IA’s indexing preference $k$

1. $\mu^1 \leftarrow U(0, L)$
2. for $p \leftarrow 1$ to $P$ do
   3. for $i \leftarrow 1$ to $N_P$ do
      4. $\{c\} \sim P$
      5. $P' \leftarrow P \cup IABART(\{c\})$
   end
   7. $P' \leftarrow I_A(P' \cup d)$
   8. Update $R(k)$ by Equation 8;
   9. Update $\mu^{p+1}$ by Equation 9;
10. return $k$

(line 8). The column probability $\mu$ is updated accordingly (line 9) for the next round.

To update the probability of sampling columns, we compute:

$$\hat{\mu}(l_j)^p = \min[\mu(l_j)^{p-1} - \alpha \frac{1}{p-1} \sum_{i < p} \hat{R}(l_j, s^i) - \beta, 0),$$

(9)

$$\mu(l_j)^p = \frac{1/(\hat{\mu}(l_j)^p)}{\sum_{j} 1/(\hat{\mu}(l_j)^p)}.$$  

We explain the details of Equation 9, 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 $\sum_{i < p} \hat{R}(l_j, s^i)$, the coefficient $\alpha$ controls how much the probability should be updated based on the new observations.

$\beta$ is a parameter to avoid exploring unwanted index columns and reduces unnecessary probing steps. 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 $\hat{R}(l_j, s^i)$ 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 cannot be optimized by building an index on $l_j$, the next probing is non-informative. Thus, in Equation 9, if the reward of a particular column $l_j$ is rarely observed in all previous rounds and thus $\mu(l_j)^p \beta - \alpha \frac{1}{p-1} \sum_{i < p} \hat{R}(l_j, s^i) > \beta$, then we use the $\min(\cdot, 0)$ function to force $\mu(l_j) = 0$.

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

5 INJECTING

Based on the indexing preference $k$ detected from the probing stage, we can generate the toxic injection workload $TW$. We only inject the toxic injection workload once, assuming the IA will re-train on the new training set after injection.

In Figure 6, $k$ can be grouped into three segments: top-ranked indexes $l_1$, mid-ranked indexes $l_2, l_3, \cdots, l_q$, and low-ranked indexes $l_{q+1}, \cdots, l_L$. 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 which will be discussed in Section 6.4.

The intuition is that $TW$ can be optimized by building indexes on columns that fall into a target segment. Thus, during the re-training, the IA will be encouraged to promote columns in the target segment and demote columns elsewhere. Clearly, toxic injection workloads should not contain queries that are optimized by top-ranked indexes. The top-ranked columns are likely to perform well on the training
workload and generate a query by injecting workloads. The stress-test will be invalid if the injection workload is broader, which helps the injection workload of the generated queries. Having more diverse queries in the injection in the IA’s search procedure to minimize its loss function. For trial-based one-off IAs, we already illustrate in Figure 4(e) that the changed ranked indexes is efficient for both one-off and trial-based IAs. For general IAs, (2) The low-ranked columns usually do not appear frequently in the target workload, meaning their stress-test 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(c), our goal is to downweight the best column (e.g., the purple star $l_4$ in the figure) and upweight the mid-ranked column (e.g., $l_3$). If after injection, the most preferred column is changed from $l_4$ to $l_3$, then, given a normal workload, the IA will likely select $l_3$, which is sub-optimal. Moreover, operating on mid-ranked indexes is efficient for both one-off and trial-based IAs. For one-off IAs, we already illustrate in Figure 4(e) that the changed model parameter will lead to degraded performance. For trial-based IAs, as shown in Figure 4(f), the stress-test will give a bad initialization in the IA’s search procedure to minimize its loss function. The IA 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 injection workload and generate a query by LAb AR T. 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 injection workload has the advantage that the column coverage of the injection workload is broader, which helps the injection workload to bypass some indexing candidate filtering heuristics.

Motivated by the above intuition, we propose the injecting procedure. As shown in Algorithm 2, the input includes query generator LAb AR T, estimated indexing preference $k = \{l_1, \cdots, l_N\}$, the dataset $d$, boundary $q$, number of columns $|\{c\}|$, toxic injection workload size $N_a$.

### Algorithm 2: Injecting procedure

**Input:** LAb ART, estimated indexing preference $k = \{l_1, \cdots, l_N\}$, the dataset $d$, boundary $q$, number of columns $|\{c\}|$, toxic injection workload size $N_a$.

**Result:** The toxic injection workload $T_W$

```plaintext
for $i \leftarrow 1$ to $N_a$
  $\{c\} \sim \{l_2, \cdots, l_q\}$
  $\hat{q} \leftarrow \text{LAb ART}(\{c\})$
  if $c(\hat{q}, d, \{c\}) < c(\hat{q}, d, l_1)$ then
    $T_W \leftarrow T_W \cup \hat{q}$
end
Return $T_W$
```

Workloads. The stress-test will be invalid if the injection 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 $l_{q+1}, \cdots, l_i$ are bad indexes, e.g., they have low index selectivity. Therefore, if we use them as the input for LAb ART, 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 toxic queries will have little impact on re-training the IA. (2) The low-ranked columns usually do not appear frequently in the target workload, meaning their stress-test 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(c), our goal is to downweight the best column (e.g., the purple star $l_4$ in the figure) and upweight the mid-ranked column (e.g., $l_3$). If after injection, the most preferred column is changed from $l_4$ to $l_3$, then, given a normal workload, the IA will likely select $l_3$, which is sub-optimal. Moreover, operating on mid-ranked indexes is efficient for both one-off and trial-based IAs. For one-off IAs, we already illustrate in Figure 4(e) that the changed model parameter will lead to degraded performance. For trial-based IAs, as shown in Figure 4(f), the stress-test will give a bad initialization in the IA’s search procedure to minimize its loss function. The IA 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 injection workload and generate a query by LAb ART. 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 injection workload has the advantage that the column coverage of the injection workload is broader, which helps the injection workload to bypass some indexing candidate filtering heuristics.

Motivated by the above intuition, we propose the injecting procedure. As shown in Algorithm 2, the input includes query generator LAb ART, estimated indexing preference $k$ and hyper-parameters such as mid-ranked index boundary $q$, number of specified columns $|\{c\}|$, toxic injection workload size $N_a$. $N_a$ toxic 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 LAb ART in Section 3 is implemented to generate an toxic query (line 3). The generated query is filtered to ensure that the top-ranked index is not the optimal index (line 4). Thus, the toxic injection 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 performance degradation of existing learning-based IAs when faced with injected extraneous workloads (Section 6.2). We then investigate the impact of hyper-parameters, including the size of the injection workload (Section 6.3), the division boundaries in the injecting strategies (Section 6.4), the number of probing epochs (Section 6.5), the hyper-parameters in Equation 9 (Section 6.6). Finally, we evaluate the performance of LAb ART (Section 6.7).

#### 6.1 Experimental Setup

**Datasets.** Since most learning-based IAs only support analytic workloads, the experiments are conducted on analytic benchmarks: TPC-H$^3$ and TPC-DS$^4$. For each benchmark, two different data sizes are generated, i.e., 1GB and 10GB.

**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 IAs cover typical learning-based IA design paradigms, except for the Monte Carlo Tree Search (MCTS) methods. We do not stress-test MCTS models in this paper because they do not have the ability to update after workload changes.

To train and retrain these IAs, 400 trajectories$^5$ (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 IA’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 IAs in total.

**Workloads.** Unless otherwise stated, to train and evaluate IAs, we follow SWIRL [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

---

$^3$[http://www.tpc.org/tpc/h/](http://www.tpc.org/tpc/h/)

$^4$[http://www.tpc.org/tpcds/](http://www.tpc.org/tpcds/)

$^5$A trajectory is a path of index selections the IA agent produces by interacting with the environment.
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 injection workload is the same as the normal workloads, i.e., $N_p = N_d = 18$ in TPC-H and $N_p = N_d = 90$ 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.** Unless otherwise stated, each experiment is repeated for 10 runs, and in each run we use Absolute performance Degradation (AD, Equation 1) and Relative performance Degradation (RD, Equation 3) to measure robustness of various IAs.

**Implementation.** The database server is a workstation with two Intel Xeon Platinum 8375C 2.90GHz CPUs and PostgreSQL 12.5. The machine learning models, including the 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.

### 6.2 Main Result

**Baselines.** We adopt five baselines and PIPA to produce injection workloads. (1) TP: each query is generated from the Templates of the target workload with a frequency that is drawn from the uniform distribution as in [19]. (2) FSM: each query is randomly generated by Finite State Machine [43] with a random seed, where each query is assigned a unit frequency. (3) I-R: Each query is generated by IABART with Randomly specified columns. (4) I-L: We used IABART to generate queries using Low-ranked columns, i.e., the bottom 50% columns in the estimated indexing preference.

(5) P-C: each query is generated by IABART using the mid-ranked columns, and the columns are ordered by the actual parameters of each index advisor. This is a clear-box variant of PIPA, and thus, it can be considered as near-optimal. (6) PIPA: the boundary of the mid-ranked index interval is $[5,1/4L]$, where $L$ is the number of columns in each dataset, and we will further explain in Section 6.4.

The effect of PIPA as a stress test. (1) Our experiments show that only PIPA and the clear-box baseline P-C always achieve positive AD on all datasets against various IAs. Other methods cannot sustain positive AD values, e.g., TP and I-R boost SWIRL’s performance by up to 40% on TPC-H 1GB after the injection (i.e., they can not cause performance degradation and they are unqualified evaluators). (2) PIPA and P-C consistently achieve the highest mean AD averaged

---

**Table 1: Relative Performance Degradation of various IAs**

<table>
<thead>
<tr>
<th>Method</th>
<th>TPC-H 1GB</th>
<th>TPC-H 10GB</th>
<th>TPC-DS 1GB</th>
<th>TPC-DS 10GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN-b</td>
<td>0.13</td>
<td>0.32</td>
<td>0.23</td>
<td>0.21</td>
</tr>
<tr>
<td>DQN-m</td>
<td>0.17</td>
<td>0.30</td>
<td>0.39</td>
<td>0.12</td>
</tr>
<tr>
<td>DRLindex-b</td>
<td>0.58</td>
<td>0.48</td>
<td>0.99</td>
<td>0.97</td>
</tr>
<tr>
<td>DRLindex-m</td>
<td>0.20</td>
<td>0.48</td>
<td>0.69</td>
<td>0.70</td>
</tr>
<tr>
<td>DBAbandit-b</td>
<td>0.32</td>
<td>0.94</td>
<td>0.41</td>
<td>0.07</td>
</tr>
<tr>
<td>DBAbandit-m</td>
<td>0.19</td>
<td>0.94</td>
<td>0.39</td>
<td>0.06</td>
</tr>
<tr>
<td>SWIRL</td>
<td>0.23</td>
<td>0.36</td>
<td>0.10</td>
<td>0.28</td>
</tr>
</tbody>
</table>

---

6.2 Main Result

**Baselines.** We adopt five baselines and PIPA to produce injection workloads. (1) TP: each query is generated from the Templates of the target workload with a frequency that is drawn from the uniform distribution as in [19]. (2) FSM: each query is randomly generated by Finite State Machine [43] with a random seed, where each query is assigned a unit frequency. (3) I-R: Each query is generated by IABART with Randomly specified columns. (4) I-L: We used IABART to generate queries using Low-ranked columns, i.e., the bottom 50% columns in the estimated indexing preference.

(5) P-C: each query is generated by IABART using the mid-ranked columns, and the columns are ordered by the actual parameters of each index advisor. This is a clear-box variant of PIPA, and thus, it can be considered as near-optimal. (6) PIPA: the boundary of the mid-ranked index interval is $[5,1/4L]$, where $L$ is the number of columns in each dataset, and we will further explain in Section 6.4.

The effect of PIPA as a stress test. (1) Our experiments show that only PIPA and the clear-box baseline P-C always achieve positive AD on all datasets against various IAs. Other methods cannot sustain positive AD values, e.g., TP and I-R boost SWIRL’s performance by up to 40% on TPC-H 1GB after the injection (i.e., they can not cause performance degradation and they are unqualified evaluators). (2) PIPA and P-C consistently achieve the highest mean AD averaged.
over runs (e.g., the orange line in Figure 7). The mean AD by PIPA is 0.2 – 0.6 higher than FS$^M$ and TP, 0.05 – 0.4 higher than I–R and I–L depending on datasets and IAs. (3) In most cases, the AD of PIPA has the least variance (e.g., narrower box in Figure 7). The above three observations indicate that PIPA is a superior stress test because it tests the IAs under abnormal conditions. It is the best opaque-box test and is comparable to the near-optimal clear-box test P–C. Other methods cannot consistently obtain extreme cases to effectively evaluate the robustness of IAs.

The effect of probing. (1) PIPA always significantly outperforms I–R, which is based on the same IABART, emphasizing the need for probing information about the IA. (2) PIPA achieves comparable results to the clear-box method P–C, meaning that the estimated ranking $k$ in Equation 5 is as effective as true model parameters in generating injection workloads. (3) We further analyze the difference between the estimated rank and the true model parameters and find that the estimated rank is sometimes more informative. For example, PIPA’s performance on DQN is even better than that of P–C. The true model parameters of the DQN model are excessively sparse, e.g., only four columns are assigned with non-zero weights, and the remaining columns are discarded under TPC-H. This brings difficulties for the evaluator to distinguish zero-weight columns and effectively generate injection workloads. In contrast, the estimated indexing preference values $K$ are denser, and a more correct list of columns can be sampled to generate the injection workloads.

The effect of IABART. The model variants, i.e., PIPA, P–C, I–R, I–L, generally outperforms other methods. 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 stress-test and have more impacts on the parameters of the IA. This phenomenon suggests that an effective test cannot 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 AD of PIPA significantly improves over I–L in most cases. Further analysis reveals 2 reasons.

Targeting mid-ranked indexes is more effective if the IA 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 stress-testing DQN and SWIRL, only “$c\_custkey$” can successfully obtain a positive AD score because “$c\_phone$” and “$o\_retailprice$” are removed by DQN’s heuristic index candidate selection and SWIRL’s invalid action masking mechanism.

Targeting mid-ranked columns traps an IA in the local optimum. We corroborate our analysis with the following two cases. The first case is shown in Fig 8(a). After stress-test by PIPA, the reward of the DQN stays around 10, which is a local optimum. After stress-test 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 stress-test 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 stress-test 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 stress-test. For example, as shown in Figure 7,
three attackers achieved $AD = 0.96$ against DRLindex-b on TPC-DS 1GB. As shown in Table 1, DRLindex-b’s RD scores are usually the highest. 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 $90 \times 432$ 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 injection 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 $1/(\text{W, d, l})$ as a reward. A small difference in the execution cost will cause the loss function to vibrate. Thus, the RD and AD scores are high if the injection workload increases the execution cost.

Performance degradation can be better mitigated 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 choice. 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 AD and RD scores are lower. (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_supp-key”/“l_part_key”. However, when SWIRL is 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 IA 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 (e.g. third row, second column of figure 7) achieve comparable AD scores with PIPA. The underlying reason is that TPC-DS contains more candidate columns, and some of the IAs are less effective in index selection. Thus, once a critical column index is knocked out, the gentle evaluator can also be sensitive, and the AD score will be the same.

6.3 Impact of Injection Workload Size

Naturally, the size of the injection workload $N_a$ has a significant impact on the evaluation results. To conveniently compare the performance among different training workloads, we fix the number of queries in the injection workload to be $N_a = 180$ and change the number of queries in the normal workload. We compute $\omega = N_a/|\text{W}|$ to measure the poisoning proportion.

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

We also report the RD result on TPC-H in Table 2. (1) PIPA yields positive RD values across different $\omega$ values, implying that PIPA is able to interfere with the training more severely than anticipated. (2) For the majority of models, the RD value increases as $\omega$ increases, which is reasonable because injecting more extraneous workloads can distort the training more significantly. (3) For DQNs, the RD reaches its maximum when $\omega = 10$. This is because DQN lacks a strong representation of workload features in its state variables. Excessively large changes in query patterns (whether gentle workload or toxic workload) can cause a sharp degradation in its performance; the two terms in Equation 3 are both large, and the RD is small.

<table>
<thead>
<tr>
<th>$\omega$</th>
<th>$10^{-2}$</th>
<th>$10^{-1}$</th>
<th>$10^0$</th>
<th>$10^1$</th>
<th>$10^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQN-b</td>
<td>0.15</td>
<td>0.19</td>
<td>0.21</td>
<td><strong>0.45</strong></td>
<td>0.02</td>
</tr>
<tr>
<td>DQN-m</td>
<td>0.32</td>
<td>0.27</td>
<td>0.42</td>
<td><strong>0.52</strong></td>
<td>0.02</td>
</tr>
<tr>
<td>DRLindex-b</td>
<td>0.08</td>
<td>0.09</td>
<td>0.22</td>
<td>0.41</td>
<td><strong>0.56</strong></td>
</tr>
<tr>
<td>DRLindex-m</td>
<td>0.11</td>
<td>0.37</td>
<td>0.08</td>
<td>0.40</td>
<td><strong>0.40</strong></td>
</tr>
<tr>
<td>DBAbandit-b</td>
<td>0.13</td>
<td>0.22</td>
<td>0.18</td>
<td>0.62</td>
<td><strong>0.63</strong></td>
</tr>
<tr>
<td>DBAbandit-m</td>
<td>0.13</td>
<td>0.22</td>
<td>0.30</td>
<td><strong>0.65</strong></td>
<td>0.61</td>
</tr>
<tr>
<td>SWIRL</td>
<td>0.04</td>
<td>0.02</td>
<td>0.27</td>
<td>0.34</td>
<td><strong>0.38</strong></td>
</tr>
</tbody>
</table>

6.4 Boundaries of the Target Segment

We investigate the impact of boundary parameters by stress-testing 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 $q = 3 \in \{2, 3, 4, 5, 6, 7\}$. We conducted five replicate experiments, and the experimental results and variance are shown in Figure 10(a).
We find that the highest AD score 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 increase AD for better stress-test effect, the target segment must exclude the best index and its 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 highest AD score is achieved at \( q = 1/4L \), yielding the highest mean AD 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 result 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 AD score decreases as \( q \) increases. When \( q = 7/8L \), the mean AD is close to 0.1 because low-ranked indexes are included, and they are ineffective for stress-test. This implies that the target segment should be placed on mid-ranked indexes.

### 6.5 Impact of 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 TPC-DS 1GB, 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 AD score improves as the number of probing epochs increases because more probing epochs lead to a more accurate estimation of the indexing preference. (2) However, only a few probing epochs are enough, e.g., \( P = 4 \) achieves the best AD score against DQN and \( P = 2 \) achieves a satisfying AD against SWIRL in TPC-H. Even in TPC-DS, a high AD can be achieved within 15 rounds of probing. (3) The probing queries are assigned with unit frequency, and running the probing stage is relatively fast, e.g., a probing round of 200 epochs for DQN on TPC-H 10GB took 1 minute for our method.

### 6.6 Impact of Parameters

In this section we investigate the impact of \( \alpha, \beta \) in Equation 9. \( \alpha \) can be seen as the learning rate that controls how much the sampling probability is updated based on new observations. It affects the ultimate stress-test performance, i.e., AD. Thus, we exponentially selected different values of \( \alpha \) and reported the AD result in Figure 12(a). Specifically, we normalized \( R(l_i, s^k) \) mentioned in Equation 9 into \([0, 1)\) and chose \( \alpha = 0.01, 0.05, 0.1, 0.5, 1, 10 \) for evaluation. We can see that the larger the value of \( \alpha \), the larger the variance of AD (i.e., the evaluator is more likely to fail), and the appropriate value of \( \alpha \) is around 0.1.

\( \beta \) acts as a sparsity term to filter out columns that are inappropriate to be index candidates, thus speeding up the estimation of indexing preference rank. Nonetheless, insufficient probing leads to inaccurate estimation. Since \( \beta \) is related to \( \alpha \) and the number of columns \( n \), we fixed \( \alpha = 0.1 \) and set \( \beta = 1/(i + n) \), \( i = 20, 10, 5, 2, 4/3 \). The reason why we consider \( i \in (1, \infty) \) is that the initial probability of each column is \( 1/n \) and the probability should always be kept positive. We reported the number of epochs required for the members of top-ranked, mid-ranked, and low-ranked indexes to stop changing in the next three epochs (blue line in Figure 12(b)). We also reported the average error rate, i.e., the average ratio of top-ranked, mid-ranked, and low-ranked indexes that are different from those under \( \beta = 0 \) (orange line in Figure 12(b)). We observe that a larger \( \beta \) leads to smaller probing rounds and larger estimation error. Thus we choose \( 1/(10 + n) \) in the main experiment to strike a trade-off between convergence speed and accuracy.
6.7 Evaluation of IABART

Evaluation Metrics. IABART is trained with 4, 326 queries 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.

$$IAC = \frac{\sum_{q^c} |q^c \cap q'\rangle |}{N(q')}, \quad (10)$$

where $q^c$ is a correct query, $q'$ is the specified index input to IABART to generate $q'$, $q'\rangle$ is the indexes selected by SWIRL for $q'$, $IAC \in [0, 1]$, 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.

$$RMSE = \sqrt{\frac{1}{N(q')} \sum_{q'} (\mathcal{R}(q') - \mathcal{R}(q^c))^2}, \quad (11)$$

where $q'$ represents a correct query, $\mathcal{R}(q')$ is the random specified reward threshold for IABART to generate $q'$, $\mathcal{R}(q^c)$ 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 $^8$.

(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. We conducted extensive experiments on GPT 3.5 turbo and GPT 4, using prompt techniques including few shot and Chain of Thought (COT). The prompts are available online $^9$. (4) IABART w/o Task x: To understand the impact of progressive training tasks, we conducted an ablation study on three variants of IABART trained without Task 1, Task 2, and Task 1 & 2, respectively.

Results. Table 3 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 and IABART w/o Task2 obtain the lowest RMSE, suggesting that Task 1 is a key training task in capturing 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.

7 RELATED WORK

Index advisors. Most conventional Index Advisors (IA) are based on heuristic algorithms that enumerate possible solutions $[6, 8, 31, 39]$. 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 $[34]$. 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. Machine learning models can be modified on the testing samples without modifying the model (i.e., evasion attack $[2]$) or contaminating the training dataset and re-training the model (i.e., poisoning attack $[41]$). Regarding the types of the victim model, poisoning attacks can be conducted against RL $[1, 23, 24, 27, 28, 36, 44]$, supervised learning $[3, 4, 15]$, and unsupervised learning models $[42]$. Existing poisoning attacks against RL can be classified into three groups, (1) manipulate the reward, environment, or action. Furthermore, PIPIA differs from

---

$^8$Specified case is on https://anonymous.4open.science/r/PIDPA-73DB/Example.md

$^9$https://anonymous.4open.science/r/PIDPA-73DB/GPT_Prompts.md

Table 3: Performance of query generation

<table>
<thead>
<tr>
<th>Method</th>
<th>GAC</th>
<th>IAC</th>
<th>RMSE</th>
<th>Distinct</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>1.00</td>
<td>0.62</td>
<td>30.41</td>
<td>0.004</td>
</tr>
<tr>
<td>DT</td>
<td>1.00</td>
<td>0.24</td>
<td>27.76</td>
<td>0.005</td>
</tr>
<tr>
<td>GPT-3.5-turbo</td>
<td>0.82</td>
<td>0.60</td>
<td>33.06</td>
<td>0.033</td>
</tr>
<tr>
<td>GPT-4</td>
<td>0.92</td>
<td>0.63</td>
<td>59.79</td>
<td>0.010</td>
</tr>
<tr>
<td>GPT-4 w/ few-shot</td>
<td>0.97</td>
<td>0.61</td>
<td>26.66</td>
<td>0.014</td>
</tr>
<tr>
<td>GPT-4 w/ few-shot &amp; COT</td>
<td>0.94</td>
<td>0.68</td>
<td>50.75</td>
<td>0.011</td>
</tr>
<tr>
<td>IABART w/o Task1&amp;2</td>
<td>1.00</td>
<td>0.58</td>
<td>37.49</td>
<td>0.030</td>
</tr>
<tr>
<td>IABART w/o Task1</td>
<td>1.00</td>
<td>0.63</td>
<td>34.77</td>
<td>0.037</td>
</tr>
<tr>
<td>IABART w/o Task2</td>
<td>1.00</td>
<td>0.62</td>
<td>25.31</td>
<td>0.040</td>
</tr>
<tr>
<td>IABART</td>
<td>1.00</td>
<td>0.73</td>
<td>26.26</td>
<td>0.043</td>
</tr>
</tbody>
</table>
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, 35], (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 to study the robustness of
updatable learning-based IAs with polluted training workloads via
PIPA. First, we proposed to probe the index preference under the
opaque-box setting. Second, we designed toxic injection workloads
to trap IAs within local optimum. Besides, we proposed a query
generation method for probing and injecting using large language
models (BART). Extensive experiments on different benchmarks
against four typical learning-based IAs demonstrated the effective-
ness of PIPA and showed the learning-based IAs are not robust
because they are easily interfered with by even a subtle amount of
extraneous injection workloads.
REFERENCES


