Master Thesis

Adversarial Workload Matters: Executing a Large-Scale Poisoning Attack against Learned Index Structures

submitted by: Matthias Bachfischer
Student ID: 1133751
Course: Master of Data Science
The University of Melbourne

supervised by: Benjamin I. P. Rubinstein
The University of Melbourne

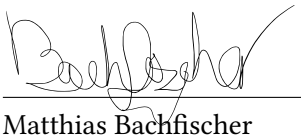
Renata Borovica-Gajic
The University of Melbourne

Melbourne, the 1st of November 2021

Declaration of Truthfulness

1. I certify that this report does not incorporate without acknowledgement any material previously submitted for a degree or diploma in any university; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person where due reference is not made in the text.
2. The report is 8000 words in length (excluding text in images, tables, bibliographies and appendices).

Melbourne, the 1st of November 2021



Matthias Bachfischer

Abstract

Databases rely on indexes to quickly locate and retrieve data that is stored on disks. While traditional database indexes use tree data structures such as B+ Trees to find the position of a given query key in the index, a learned index structure considers this problem as a prediction task and uses a machine learning model to "predict" the position of the query key. This novel approach of implementing database indexes has inspired a surge of recent research aimed at studying the effectiveness of learned index structures. However, while the main advantage of learned index structures is their ability to adjust to the data via their underlying ML model, this also carries the risk of exploitation by a malicious adversary.

In this work, the results from executing a large-scale poisoning attack on dynamic learned index structures are presented. The poisoning attack used in this research targets linear regression models and works by manipulating the cumulative distribution function (CDF) on which the model is trained. The attack deteriorates the fit of the underlying ML model by injecting a set of poisoning keys into the dataset, which leads to an increase in the prediction error of the model and thus deteriorates the overall performance of the learned index structure.

The effectiveness of the poisoning attack described in this research is evaluated against three index implementations by measuring their throughput in million operations per second. The evaluated indexes consist of two learned index structures ALEX and Dynamic-PGM as well as a traditional B+ tree. For each index, its performance on a variety of real-world datasets and workload scenarios is measured. The experimental results show that learned index structures can exhibit a performance deterioration of up to 20% when evaluated on the poisoned vs. non-poisoned datasets. Contrary to that, the B+ tree does not exhibit any performance deterioration when evaluated on the poisoned datasets. This shows that learned index structures are not robust to adversarial workload and can be manipulated by an adversary to achieve significant slow-down compared to traditional data structures.

Contents

Chapter

1

Introduction

A database index is a data structure that is widely used in databases to organize data for fast retrieval. Using an index allows the database to quickly locate the data without having to first search through all entries in the database each time a specific record needs to be retrieved.

In traditional database design, tree-based data structures such as B+ trees and its variants have seen wide adoption due to their relative ease of implementation and optimal worst-case guarantees in terms of algorithmic complexity. A B+ tree is self-balancing data structure that maintains the data in a sorted format and allows searches, insertions, and deletions with constant time complexity $O(\log n)$ (where n is the total number of elements in the B+ tree). It is a general-purpose data structure that makes no prior assumptions about the data distribution and does not take advantage of any patterns that might be specific to the data that it stores.

In practice, real-world data often follows some underlying pattern that, if modeled correctly, could significantly speed-up the data retrieval process. For example, given a set of continuous integer keys (keys from 1 to 100 million), one would not construct a conventional B+ tree index over the keys. Instead, the key itself could be used as an offset for the index, thus reducing the time required to look-up any key in the dataset from $O(\log n)$ to $O(1)$. This increase in performance is what Kraska et al. hoped to achieve when they first introduced their work on *Learned Index Structure* (LIS) models [1]. Even though the concept of a LIS is relatively novel, it has already led to a surge of results that leverage ideas from *Machine Learning* (ML), data structures, and database systems [2], [3], [4], [5], [6], [7], [8], [9], [10],[11], [12], [13], [14], [15], [16].

The core idea of a LIS is to model the functionality of a data structure as a prediction task, i.e. given an input key, the objective of the LIS is to predict the key's position in a sorted collection of key-value pairs. This approach allows the use of continuous functions to encode the data and leverage learning algorithms to approximate the function. Specifically, the approach proposed by Kraska et al. [1] is to approximate the *Cumulative Distribution Function* (CDF) of the keys. Given a key k as an input, the CDF returns the probability that the chosen key takes a value less than or equal to k (i.e. $P(X \leq \text{Key k})$).

Based on this observation, one can use the CDF to:

(1) compute the number of keys less than the (queried) key k and

(2) infer the key's memory location (assuming the keys were sorted during the initialization).

In the context of learned indexes, the term CDF is frequently used synonymously to describe a mapping from a key to its position in the sorted array. This is contrary to the statistical definition

of the CDF as the probability that a random variable will take a value less than or equal to a given key. This work uses the former interpretation of the CDF as a mapping from the key to its position in an sorted array.

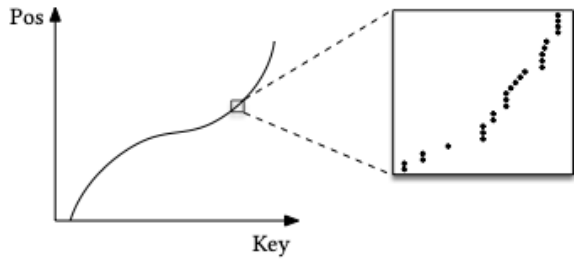


Figure 1: Learned index as a CDF [1]

Figure 1 shows that the CDF primarily describes a local linear regression problem that is monotonic increasing in the covariate (the key). The CDF gives an approximate location - that it sums to one is not strictly required for this scenario. Furthermore, in order to model larger datasets, multiple local linear regression models may be used to describe different parts of the CDF.

In the work of Kraska et al. [1], a linear regression on the CDF is one of the essential building blocks that has been shown to work well and can be combined with hierarchical *Recursive Model Index* (RMI) structures to balance the resulting model with respect to latency, memory usage, and computational cost. Using linear regression to construct a LIS is beneficial, because it only requires the storage of relatively small number of parameters (slope a and intercept b) and performing prediction only involves basic operations for multiplication and addition.

The hierarchy of the LIS provides great flexibility because it represents a “mixture of experts”, where each expert is responsible for subsets of the data. As an example, the upper levels of the hierarchy can be a model that approximates well the general shape of the function (e.g., a neural network model), while the next level consists of fast and low-memory models which are able to capture the local structure well (e.g., linear regression model).

Chapter

2

Related Work

2.1 Learned Index Structures

2.1.1 Static Learned Index Structures

Since the first published work on learned index structures by Kraska et al. [1], the research community has explored a variety of ideas on how LIS could be used as a replacement for traditional index structures such as B+ trees. In the following sections, an overview of the most prominent LIS models and their core characteristics is given.

Recursive Model Indexes (RMI)

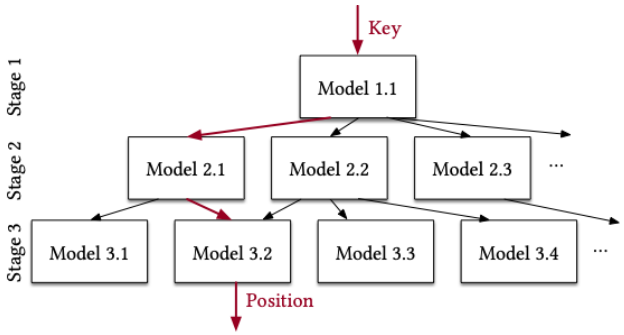


Figure 2: A multi-staged RMI model [1].

The RMI model was initially presented by Kraska et. al in [1] and has since then ignited the recent surge of research in the area of learned index models. The RMI uses a hierarchy of ML models that are organized as a *Directed Acyclic Graph* (DAG). It is trained to learn the input distribution of the pairs (k, rank(k)) for all $k \in K$, with K denoting the underlying keyset. At query time, each model (starting from the top one) takes the query key k as input and picks the following model in the DAG that is "responsible" for that key (as shown in Figure 2). The output of the RMI is the approximate position of the key returned by the last queried ML model. A binary search is subsequently executed within a range of neighbouring positions of the predicted position, with the size of the binary search range depending on the prediction error of the RMI.

In the case of a two-stage RMI, a function F is trained on N data points (key, $index$) pairs. The RMI F is composed of a single first-stage model f_1, and B second-stage models f_2^i where the value B denotes the "branching factor'" of the RMI.

Formally, the two-stage RMI is defined as

$$F(x) = f_2^{\lfloor B \times f_1(x)/N \rfloor}(k) \tag{2.1}$$

and uses the stage-one model $f_1(x)$ to compute an approximation of the CDF of the input key k. Next, this approximated key position is scaled to a range between 0 and the branching factor B. The scaled value is used to select a model from the second stage $f_2^i(k)$ which is then used to produce the final approximation. In the RMI, the first-stage model $f_1(k)$ can be thought of as partitioning the data into B buckets, and each second-stage model $f_2^i(k)$ is responsible for approximating the CDF of only the keys that fall into the i-th bucket.

One could assume that ML models cannot provide the guarantees ensured by traditional indexes, both because they can fail to learn the distribution and because they can be expensive to evaluate. However, in the experiments performed by [1], the RMI dominated the B+ Tree and was up to 1.5-3× faster while requiring two orders of magnitude less storage space.

Radix Spline Indexes (RS)

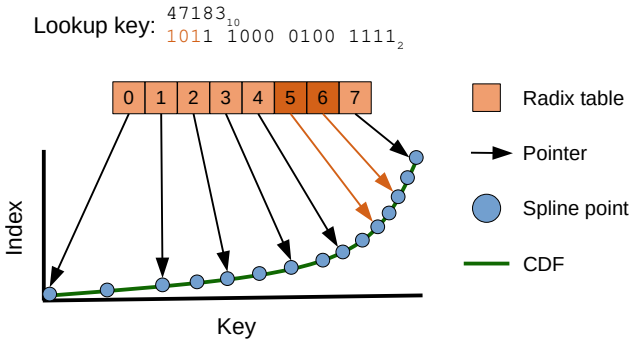


Figure 3: A Radix Spline index [13].

The *Radix Spline* (RS) index is another LIS introduced in [13]. It consists of a linear spline that approximates the CDF of the data and a radix table that indexes the spline points. The RS is built from the data in a bottom-up fashion and can be constructed in a single pass with a constant worst-case cost per element.

Figure 3 shows that the radix table indexes r-bit prefixes of the spline points and serves as an approximate index to accelerate binary searches over the spline points. Internally, it is represented as an array containing 2^r offsets into the sorted array of spline points. The spline points themselves are represented as $(key, index)$ pairs. To locate a key in a spline segment, linear interpolation between the two spline points is used.

Piecewise Geometric Model Indexes (PGM)

The *Piecewise Geometric Model* (PGM)-Index [12] is a multi-level structure, where each level represents a *Piecewise Linear Approximation* (PLA) bounded by an error ϵ.

Figure 4 depicts the construction of an example PGM-Index. In the first level, the data is partitioned into three segments, each segment represented by a simple linear model (f_1, f_2, f_3). Each of the

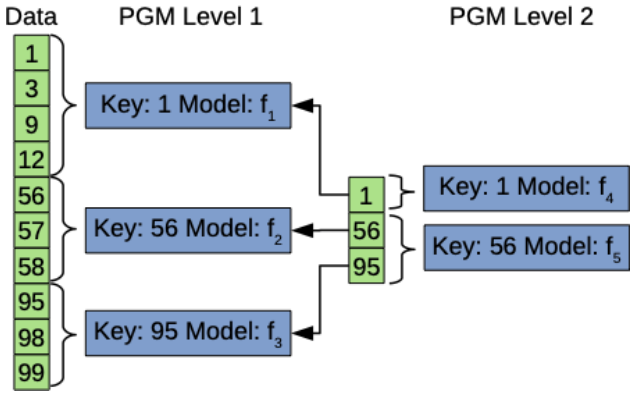


Figure 4: A PGM index [12].

linear models is responsible for modeling the CDF of the keys in their corresponding segments within a preset error bound ϵ. In the next level, the partition boundaries of the first level models are treated as their own sorted dataset, and another error-bounded piecewise linear regression is computed. This is repeated until the top level of the PGM becomes sufficiently small.

Formally, the PLA of the PGM-Index computes a piecewise linear regression and partitions the data into $n + 1$ segments with a set of points $p_0, p_1, \ldots, p_n$. The entire piecewise linear regression is expressed as a piecewise function:

$$F(x) = \begin{cases} a_0 \times x + b_0 & \text{if } x < p_0 \\ a_1 \times x + b_1 & \text{if } x \geq p_0 \text{ and } x < p_1 \\ a_2 \times x + b_2 & \text{if } x \geq p_1 \text{ and } x < p_2 \\ \ldots & \\ a_n \times x + b_n & \text{if } x \geq p_n \text{ and } x < p_n \end{cases}$$

Each regression in the PGM index is constructed with a fixed error bound ϵ and yields in a regression that can be used as an approximate index.

2.1.2 Dynamic Learned Index Structures

A natural evolution of the static LIS models introduced in the section above is to extend them to also handle dynamic updates. In 2019, Hadian and Heinis were the first to investigate approaches for building "update-adaptive models" [17] by minimizing the error that the models incur when inserting and deleting data. *Adaptive Learned indEX* (ALEX), a LIS published in 2020 [2] was the first open-source LIS implementation that natively supported updates. The PGM-Index discussed in Section 2.1.1 also supports insert and delete operations via the Dynamic-PGM-Index. *Learned Index with Precise Positions* (LIPP) [18] and *Cache-cOnscious Learned INdex* (COLIN) [19] (both published in 2021) are the two most recent, updatable data structures and are also briefly described below.

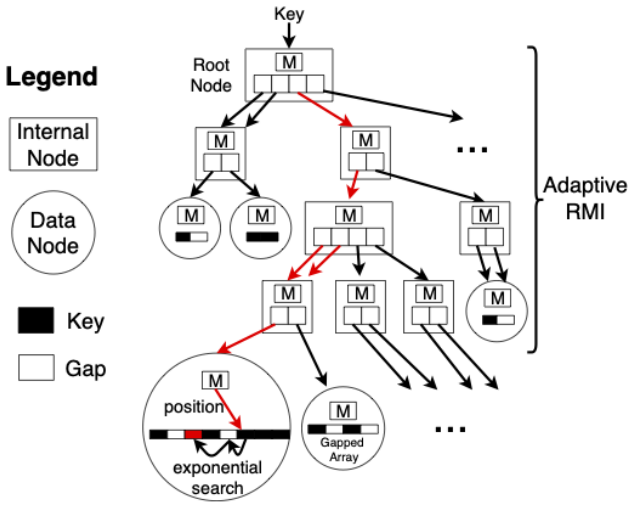


Figure 5: The design of ALEX [2].

Adaptive Learned indEX (ALEX)

The ALEX design shown in Figure 5 builds on the RMI model published in [1]. However, while the RMI is of static nature and uses a fixed number of levels and fixed number of models in each level, ALEX extends the original RMI design with respect to two key improvements.

The first major improvement of ALEX is that it allows dynamic updates to the data by using a RMI model based on PLA. Depending on the workload, ALEX can dynamically adapt the structure of the RMI to yield a better fit for the data. This is done via linear cost models that predict the latency of lookup and insert operations based on statistics measured from a RMI.

The second major improvement is that ALEX, similar to a B+ Tree, stores its data at the leaf level. This means that every leaf node stores an array of keys and payloads, thus allowing the data structure to grow dynamically while limiting the number of expand and shift operations that are required. As an improvement over a traditional B+ Tree, ALEX makes use of a special *Gapped Array* (GA) layout for the leaf nodes, which allows it to achieve faster insert and lookup times compared to traditional data structures.

Dynamic Piecewise Geometric Model Indexes (Dynamic PGM)

The Dynamic PGM-Index is another LIS implementation with support for dynamic updates. It is an extension to the PGM-Index described in Section 2.1.1. In this section, a brief overview of the core characteristics of the Dynamic PGM-Index for handling updates will be given [12]. The dynamic PGM-Index is constructed on a series of (static) PGM-Indexes built over sets $S_0, ..., S_b$ which are either empty or have key size $2^0, 2^1, ..., 2^b$, where $b = \Theta(\log n)$. When inserting a new key k into the dynamic PGM-Index, the first empty set S_i in the data structure is retrieved. In the next step, a new PGM-Index over the merged set $S_0 \cup S_{i-1} \cup \{k\}$ is constructed, where the new, merged set is then used as S_i, and the previous sets are emptied. The deletion of a key k is handled similarly to an insert operation by adding a special tombstone value which indicates the logical removal of d [20].

Updatable Learned Index with Precise Positions (LIPP)

LIPP is the most recent learned index model published by Wu et al. in [18]. Similar to other learned index implementations, LIPP works by combining a series of models in a recursive layout where each model is responsible for a certain subset of the data (as shown in Figure 6). To find the correct model for a given key, LIPP uses the *Fastest Minimum Conflict Degree* (FMCD) algorithm that helps to find the optimal model.

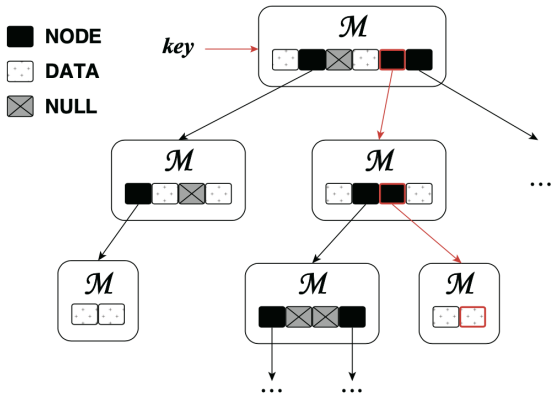


Figure 6: The structure of LIPP [18].

Cache-cOnscious Learned INdex (COLIN)

COLIN (Cache-cOnscious Learned INdex) [19] is another learned index model that was published in 2021. It uses a model-based data placement policy and cache-conscious data layout to optimize the retrieval of data. However, because COLIN is not available as open-source code to the research community and does not introduce any novel approaches compared to the existing learned indexes, it was not studied in further detail in this work.

2.1.3 Discussion

From the description of LIS implementations like RMI, RS and PGM-Index, it can be noted that all indexes work by approximating the CDF of the underlying data. However, all of the learned index structures differ in a variety of aspect, most notably (1) the type of model used to approximate the CDF, (2) whether the index supports updates and (3) whether the implementation is available as open-source. Table 1 gives an overview of the most prominent static and dynamic LIS models.

The main types of models that are used by learned index structures are the *Piecewise Linear Approximation* (PLA) and *Linear Spline* (LS) model. The PLA model is a variant of the linear regression model that tries to approximate the CDF of the dataset by diving it into variable-sized segments. The first and last keys of each segment are then used to construct a linear approximation of the data, resulting in a PLA model. In contrast to that, the LS model fits the data by approximating the CDF via linear spline points.

From Table 1, we can observe that the majority of indexes use only a single type of model in constructing the learned index: linear spline regression for RS and PLA for PGM-Index and ALEX.

Index	Model	Updates	Open-Source
RMI	Multiple	✗	✓
RadixSpline	LS	✗	✓
PGM-Index	PLA	✗	✓
ALEX	PLA	✓	✓
Dynamic-PGM-Index	PLA	✓	✓
LIPP	PLA	✓	✓
COLIN	PLA	✓	✗

Table 1: Overview of LIS implementations

The RMI is the only LIS that supports a variety of model types, which gives the RMI a greater degree of flexibility, but also increases the complexity of tuning the RMI (as shown in [21]).

With respect to dynamic (updatable) learned index structures, there are two key implementations that are subject to active investigation by the community. One is ALEX, a dynamic learned index structure that builds on the RMI architecture. The second is the Dynamic-PGM-Index, the dynamic implementation of the PGM-Index. Both ALEX and PGM-Index use a PLA model (a variant of linear regression model) and are available as open-source, which makes them an ideal target for mounting a poisoning attack against LIS models. The unique characteristics of the two data structures as well as their worst-case complexities is further discussed in Section 4.1.

2.2 Adversarial Machine Learning

2.2.1 Poisoning Attacks in Adversarial ML

The term Adversarial Machine Learning describes the study of *Machine Learning* (ML) techniques against an adversarial opponent that aims at fooling a model by supplying deceptive input. Adversarial ML has emerged in recent years as a new field, mostly driven by new advancements in computing capabilities [22], [23].

Within Adversarial ML, data poisoning attacks have a long history and have been applied in a variety of contexts such as poisoning of neural network or recommender systems [24], [25]. To this date, research on data poisoning has mostly been focused on classification and anomaly detection [26], [27], [28], while adversarial regression has largely remained underrepresented [29].

In 2015, Xiao et al. first introduced a gradient-based optimization framework for linear classifiers like Lasso or Ridge Regression [30]. Jagielski et al. [31] have built upon this work and extended their approach for linear regression while also proposing a defense mechanism against poisoning attacks called *TRIM*.

In 2020, another novel attack algorithm on regression learning was proposed by Müller et al. [32]. It works by manipulating the training dataset in a way that causes maximum disturbance of the data points. In their experimental evaluation, the authors were able to observe that the *Mean Squared Error* (MSE) of the regressor increased by 150 percent after inserting only 2% of

poisoned samples.

Poisoning attacks on LIS models differ significantly from previous attempts of poisoning linear regression models, because it requires poisoning of the CDF. This is challenging because every insertion affects the values of all points of the dataset. This "compound effect of poisoning" is described further in Section 4.1.

2.2.2 Poisoning Attacks on Learned Index Structures

The study of poisoning attacks on LIS is a new area of research first studied by Kornaropoulos et al. in [33]. In the context of a LIS, the aim of a data poisoning attack is to inject a set of maliciously crafted data into the models and thus cause inaccurate predictions on the location of legitimate data.

To investigate whether the performance of learned index structures can be manipulated via poisoning attacks, the researchers in [33] have proposed two poisoning attacks on the hierarchical architecture of the RMI model:

(1) as multiple-point poisoning attack on the CDF that finds the optimal set of poisoning keys which maximize the poisoning effect on the loss function (*GreedyPoisoningRegressionCDF*)

(2) a poisoning attack for the two-stage RMI model which leverages the multiple-point poisoning attack (1) such that it increases the error of the overall model (*GreedyPoisoningRMI*)

The poisoning attack by Kornaropoulos et al. [33] essentially works by confusing the model so that it can no longer correctly decide in which of two neighboring second-stage models the key resides and hence causes the RMI to perform an expensive binary search over a wide range of keys. Using this approach, the researchers were able to increase the error of the poisoned RMI model by a factor of up to 300× compared to a non-poisoned model.

Chapter

3

Preliminaries

3.1 Terminology

In this thesis, a key is denoted by k and its key universe (range of potential keys) as $\mathcal{K}$, where $|\mathcal{K}| = m$. The set of all keys of an index is denoted as $K \subseteq \mathcal{K}$. The *density* of a keyset K is calculated via the ratio $|K|/|\mathcal{K}| = n/m$. Similar to previous work on LIS models, it is assumed that keys are given as non-negative integer and that the total order of the keyset can always be derived (similar to the original work on LIS models in [1]). It is further assumed that each key is associated with a record and that the records are stored in an in-memory array that is sorted with respect to the key values.

3.2 Background on Poisoning Attacks

This research focuses on poisoning attacks against dynamic LIS models that make use of *Piecewise Linear Approximation* (PLA). Let $D = \{x_i, y_i\}_{i=1}^{n}$ denote the data used by a learned index structure, with $x \in \mathbb{R}^d$ representing the input data vector and $y \in \mathbb{R}$ representing the output variable. In an ordinary linear regression model, the output is computed via a linear function $f(x, a, b) = a^T x + b$ with parameters $a \in \mathbb{R}^d$ and $b \in \mathbb{R}$. The parameters a, b are chosen so as to minimize the loss function $\mathcal{L}(D, a, b) = \frac{1}{n} \sum_{i=1}^{n} (f(x_i, a, b) - y_i)^2$, also known as the Mean Squared Error.

To this date, previous work on poisoning attacks have either focused on *gradient-based* poisoning attacks or investigated alternatives to analytically solve the optimization problem described above, for example by using a *sampling-based* [31] or *generative* approach [34].

The poisoning attack described in this work takes a different approach: It exploits the structure of the CDFs to compute the location of the poisoning point that maximizes the error when regressing on the legitimate keys. It builds upon the *GreedyPoisoningRegressionCDF* algorithm proposed by Kornaropoulos et al. [33], which has been significantly extended to allow it to scale to very large datasets with > 200M keys.

3.3 Adversarial Model

This section described the adversarial model for poisoning attacks against LIS. It is inspired by previous research on poisoning attacks [35], [22], [31], and consists of a definition of the adversary's goals, the adversary's capabilities & knowledge as well as the evaluation metric that is used to measure the effectiveness of the poisoning attack on LIS.

3.3.1 Adversary's Goals

When executing a poisoning attack against a LIS model, the adversary's goal is to corrupt the learned index model during the training phase (i.e.,while the index is being constructed), so that its performance deteriorates during the test phase (i.e., when the index is used to predict the position of a given key).

Poisoning attacks can be categorized into two main categories: *poisoning integrity attacks* and *poisoning availability attacks*. In *poisoning integrity attacks*, the adversary tries to cause specific mispredictions on certain datapoints during test time, while in *poisoning availability attacks*, the adversary's goal is to affect the prediction results indiscriminately and thereby deteriorate the overall performance of the model.

The focus of this research is on *poisoning availability attacks*, where the adversary's goal is to deteriorate the perceived performance benefit of a learning-based data structure compared to a more traditional, tree-based data structures. Specifically, the objective of the adversary is to generate a small number of *poisoning keys* that are used to augment the training dataset that consists of so-called *legitimate keys*. The assumption is that training a LIS model with both the *poisoning keys* and *legitimate keys* will result in a model whose performance is worse compared to a LIS model trained on only the *legitimate keys*.

3.3.2 Adversary's Capabilities & Knowledge

Data poisoning attacks distinguish between two attack scenarios, depending on the adversary's capabilities and knowledge: *white-box* and *black-box* poisoning attacks.

In *white-box attacks*, the attacker is assumed to have full access to the training data, i.e., the keyset K and the slope and intercept parameters a and b of the linear regression. White-box attacks have already been executed previously with great success in a variety of settings [36], [26] and have also proven to work well in the context of LIS models [33].

Contrary to that, *black-box attacks* assume that the adversary has no direct access to the training data or training parameters of the model. In this scenario, the adversary first needs to infer the parameters of the model and use his estimates to perform the poisoning attack. Though more difficult to execute, black-box attacks allow better transferability of poisoning attacks against different training sets, as shown in [30] and [25].

The focus of this work is to study white-box poisoning attacks. When performing the poisoning attack, it is assumed that the attacker is able to inject up to p maliciously-crafted poisoning keys into the training set prior to training the LIS model. Figure 7 illustrates the execution of the poisoning attack. The total number of data points in the training set is given by $N = n + p$, where n denotes the number of legitimate keys and p the number of poisoning keys in the training data. Similar to previous work, it is assumed that the adversary is able to control only a tiny fraction of the training set limited by the poisoning percentage α, where $\alpha = p/n$ [31], [30].

To extend the poisoning attack to black-box scenarios in which the adversary only has access to the underlying data distribution (or another dataset generated from the same source as the training data), the attacker would first need to infer the parameters of the underlying PLA model. However, because the choice of parameters of the learned index structures is limited and depends heavily on the specific model architecture, it should be relatively easy to infer the parameter

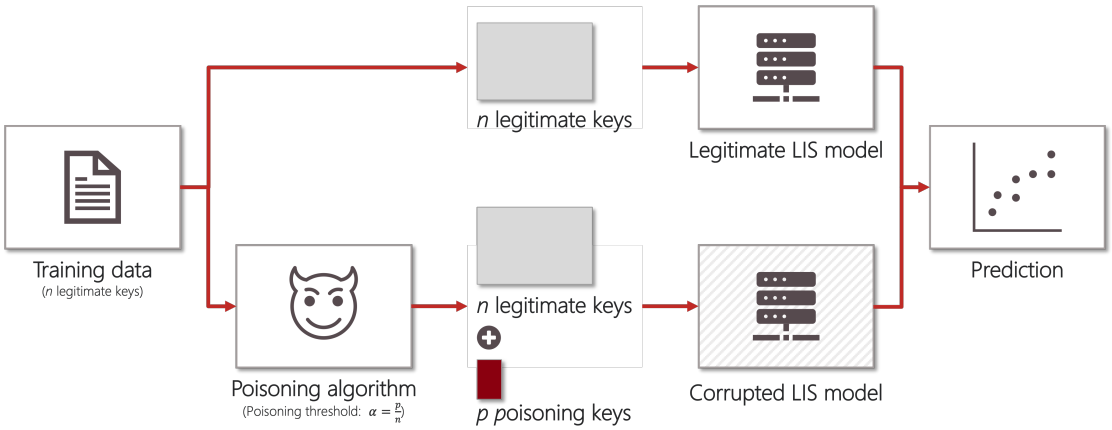


Figure 7: Execution of data poisoning attack.

of the models when executing the poisoning attack in a black-box setting. Once the adversary has successfully estimated the parameters of the LIS model, he can continue with the white-box poisoning attack as described here.

3.3.3 Attack Evaluation Metric

In the original work on LIS, the performance of the RMI is evaluated by measuring the lookup time in nanoseconds [1]. Since then, a variety of efforts have been undertaken to facilitate the comparison between different learned indexes with respect to their predictive accuracy, lookup times and index build times [37], [38], [39].

In this research, the effect of poisoning the LIS is measured with respect to the throughput in million operations per second of a LIS model trained on the non-poisoned dataset and a LIS model trained on the poisoned dataset. The throughput is measured against a variety of workloads further described in Section 6.1.2. Also, for each workload and dataset, the performance deterioration between a model trained on the legitimate data and a model trained on the poisoned data is observed.

Chapter

4

Poisoning Attacks on Dynamic Learned Index Structures

4.1 Poisoning of Learned Index Structures by Manipulating the CDF

Executing a poisoning attack on learned index structures can be based on two potential scenarios: Poisoning the LIS by manipulating the CDF of the data (this section), and poisoning the LIS by injecting extreme outlier keys into the dataset (next section).

When a LIS is poisoned by manipulating the CDF, the aim of the attacker is to perturb the underlying data distribution on which a LIS is trained in such a way that it becomes difficult to approximate. As shown in Chapter 2, the majority of LIS models rely on linear regression models to approximate the CDF [1] [21] and even though more complex approaches like neural networks or logarithmic error regression have proven to also work well [40], none of those models support dynamic updates yet. This research therefore focuses on the case of linear regression models.

In linear regression models, an ideal poisoning attack should aim at making the underlying data distribution of the CDF non-linear, thus deteriorating the fit of the linear regression and increasing the error of the LIS during key lookup. Although the algorithmic complexity of a LIS can be $O(1)$ in the ideal case, the worst-case complexity of a LIS may be significantly worse than that. Table 2 shows the worst-case complexity for lookups, inserts and deletes for the three data structures ALEX, B+ tree and PGM-Index.

	ALEX	**B+ tree**	**PGM-Index**
Lookup Complexity	$O(\log N + \log m)$	$O(\log N)$	$O(\log^2 N)$
Insert Complexity	$O(\log^2 N + \log m)$	$O(\log N)$	$O(\log^2 N + \log N)$
Search Range	$O(m)$	$O(m)$	$O(\epsilon)$

N = number of keys, m = maximum size of leaf node, ϵ = error bound in PGM-Index

Table 2: Worst-case complexity for lookups & inserts among dynamic Learned Indexes.

The ALEX data structure [2] uses a PLA based on linear regression when storing records in its *Data Nodes*. In the worst-case, a lookup operation in ALEX can cost up to $O(\log N + \log m)$, contrary to $O(\log 1)$ in the best-case. The PGM-Index [12] is another LIS implementation that relies on a PLA model. Similar to ALEX, the PLA model in the PGM-Index is used to perform a mapping between the keys and their approximate positions in the sorted array. Because the

dynamic version of the PGM-Index works by separating the keys into subsets with different sizes and building individual PGM indexes over those subsets, its' worst-case lookup performance is $O(\log^2 N)$, compared to $O(\log 1)$ in the best-case.

Given this context, it should be noted that tree-based data structures like B+ trees do not rely on modeling the CDF, but instead rely on the key value to retrieve the data from the tree. Therefore, the B+ tree data structure and its variants should not be vulnerable to a poisoning attack on the CDF.

4.2 Poisoning of Learned Index Structures by Introducing Outlier Keys

Another approach to poison the LIS is to introduce extreme outlier keys that exploit the internal structure of the LIS implementation to deteriorate the performance. Because such an attack would be trivial to detect and rectify by simply scanning the keyset for extreme outliers, it has not been investigated in-depth as part of this research.

However, while performing some experiments related to the CDF poisoning approach described in Section 4.1, the author of this work observed that the ALEX implementation by Microsoft [41] exhibits very poor performance in the presence of extreme outlier keys. This is likely caused by the fact that introducing extreme outlier keys in the dataset will result in ALEX's key domain and tree depth to become unnecessarily large. In some circumstances, this can lead to a crash of ALEX. Figure 8 shows the insertion of several `uint64` keys with key value $2^{64} - 1$, which triggers a crash with error message `std::bad_alloc`.

A potential poisoning attack deliberately targeting the ALEX implementation could exploit this condition by introducing extreme outlier keys during the poisoning process. A potential mitigations for this attack would be to add special logic for handling extreme outliers, or to have a modeling strategy that is more robust to sparse key spaces.

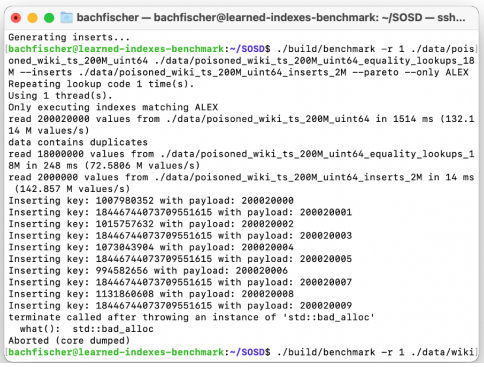


Figure 8: Introducing extreme outlier keys can trigger a crash in ALEX with < 10 poisoning keys.

Chapter

5

Poisoning of Linear Regression Models by Manipulating the CDF

5.1 Theoretical Setup

This section describes the theoretical setup for a poisoning attack on linear regression models by attacking the CDF of the training data. The attack works by inserting a certain number of *poisoning keys* into the training dataset with the aim of increasing the approximation error of the regression and thus deteriorate the overall performance of the index. The attack is based on the observation that LIS models like RMI or ALEX work by approximating the relative order of a key-value pair, where the value denotes the rank of the queried key. If the predicted approximation of the rank is accurate, it allows the LIS to directly retrieve the key-value pair from memory without having to perform searches on the rest of the data (assuming that the key-value pairs are stored in ordered manner).

A LIS consists of an index that is being constructed on a keyset K of size n, where each key $k \in K$ has a rank r in the interval $[1, n]$. Here, r denotes the position of k in an ordered sequence of K. The objective of the LIS is to approximate the rank of the queried key by constructing a regression model on (k, r), where the X-value is given by the key k, and the Y-value is denoted by the rank r. In other words, the function that the regression model approximates is the CDF of the input dataset.

Prior work on poisoning attacks on linear regression models was aimed at inserting maliciously-crafted poisoning keys that cause a "local change", i.e., inserting keys that do not affect the X- and Y-values of the legitimate points [31]. In the case of LIS models, the insertion of a single, maliciously-crafted key $k_{poisoned}$ will cause a shift in the rank of all keys larger than $k_{poisoned}$. This change will in turn trigger a shift of the CDF, thus compounding the effect of the adversarial insertion. To this date, the "compounding effect of adversarial insertion" has only been studied by the authors of [33]. This thesis will build up on their results and investigate the effects of poisoning attacks on dynamic learned index structures such as ALEX and Dynamic-PGM.

5.2 Linear Regression Models on CDF

To fit a linear regression model on CDF, LIS models usually rely on the ordinary least squares method as described in Definition 1.

Definition 1 (Linear Regression on CDF). *Let $K = \{k_1, \cdots, k_n\} \subseteq \mathcal{K}$ be the set of integers that correspond to the keys of the index. Every key $k_i \in K$ has its associated rank $r_i \in [1, n]$. The linear regression model on a CDF computes a pair of regression parameters (a, b) that minimizes the following mean squared error (MSE) function :*

$$\min_{a,b} \mathcal{L}\left(\{k_i, r_i\}_{i=1}^n, a, b\right) = \min_{a,b}\left(\sum_{i=1}^n (ak_i + b - r_i)^2\right).$$

The minimization problem of Definition 1 can be solved by deriving a closed-form solution when the set K is treated as a sample from the set of keys $\mathcal{K}$. Based on this, the *sample mean* of the key set is defined as M_K and the sample mean of the rank set is defined as M_R. Similarly, the *sample variance* is defined as Var_K and Var_R, and the *sample covariance* between K and R as Cov_{KR}. Lastly, the sample mean of the squares of the keys, resp. ranks, is defined as M_{K^2}, resp. M_{R^2}. Recall that the formulas of variance and covariance are $\mathrm{Cov}_{XY} = M_{XY} - M_X M_Y$, $\mathrm{Var}_X = M_{X^2} - M_X^2$ and $\mathrm{Var}_X = \mathrm{Cov}_{XX}$.

Using these formulas and the fact that the rank R can be approximated via $R = aK + b$ gives us $\mathrm{Cov}_{KR} = Cov(K, aK + b)$. To solve these equations for a and b, we use $\mathrm{Cov}_{KR} = a \cdot \mathrm{Cov}_{KK} + 0 = a \cdot \mathrm{Var}_{KK}$ which yields $a = \frac{\mathrm{Cov}_{KR}}{\mathrm{Var}_K}$ and $b = M_R - a^* M_K$.

The Linear Regression from Definition 1 hence admits the following closed-form solution:

$$a^* = \frac{\mathrm{Cov}_{KR}}{\mathrm{Var}_K}, \quad b^* = M_R - a^* M_K.$$

5.3 Poisoning Attacks on CDFs

In [33], the authors introduced a novel poisoning attack for linear regression on CDFs called *GreedyPoisoningRegressionCDF*. However, to generate a poisoning key, the *GreedyPoisoningRegressionCDF* algorithm requires one iteration through the whole keyspace K. Because of the massive amount of computation involved, the algorithm therefore does not scale to the context of very large datasets (> 200M keys). This research addresses these shortcomings by introducing a distributed poisoning attack *GreedyDistributedPoisoningRegressionCDF* that can be scaled to use an arbitrary number of computational resources.

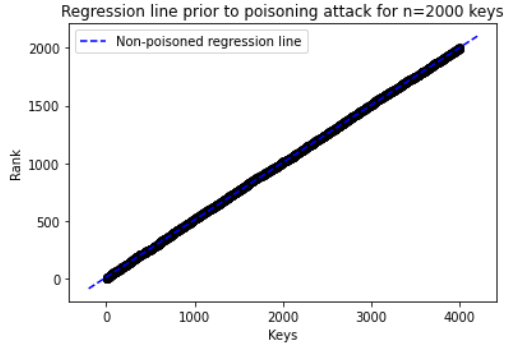
Definition 2 describes the poisoning strategy of an adversary targeting linear regression models on CDF. The parameter λ denotes the upper bound that limits the size of the poisoning keyset P and is chosen to be proportional to the size of the keyset. In the experiments described in Section 6.1, λ was set to $\lambda = 0.0001n$.

Definition 2 (Poisoning Linear Regression on CDF). *Let K be the set of n integers that correspond to the keys and let P be the set of p integers that comprise the poisoning keys. The augmented set on which the linear regression model is trained is $\{(k'_1, r'_1), (k'_2, r'_2), \cdots, (k'_{n'}, r'_{n'})\}$, where $k'_i \in K \cup P$ and $r'_i \in [1, n+p]$. The goal of the adversary is to choose a set P of size at most λ so as to maximize the loss function of the augmented set $K \cup P$:*

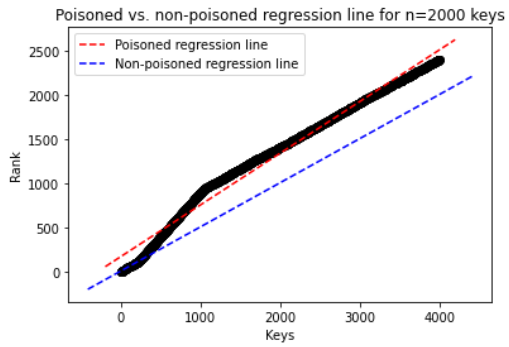
$$\arg\max_{P \text{ s.t. } |P| \leq \lambda} \left(\min_{a,b} \mathcal{L}\left(\{k'_i, r'_i\}_{i=1}^{n+p}, a, b \right) \right)$$

5.3.1 The Effect of Poisoning on CDF

This section describes the effect of the poisoning attack on CDF by using a simple example with $n = 2000$ keys. Figure 9 (a) shows the regression line for the original key set K on the X-axis and the corresponding ranks R on the Y-axis, while Figure 9 (b) shows the regression line after the poisoning. To calculate the MSE, one can sum up the squares of the distances of the points from the regression line.



(a) Regression before poisoning (MSE = 22.99).



(b) Regression after poisoning (MSE = 8675.83).

Figure 9: Illustration of the compound effect of poisoning.

In a typical setting of poisoning regression models, the addition of a single point has limited overall impact since all the other points stay in their original X-, Y-coordinates. Thus, even if the original regression line is maintained, the MSE error would only increase by the contribution introduced by the individual poisoning point. In the case of poisoning on CDF, the addition of a single point can affect the rank (i.e., Y-coordinate) of many original points of the CDF.
This is known as the *compound effect of poisoning on CDF*: Inserting p poisoning keys at location $k_{insertion}$ will shift the ranks of the points with key $k > k_{insertion}$ upwards by the number of poisoning keys p that were inserted.

5.3.2 GreedyPoisoningRegressionCDF Algorithm

A poisoning attack that specifically targets the CDF has been first introduced by the authors of [33]. In their work, the authors derived a closed-form solution to compute the evaluation of the loss function for the poisoned keyset. In the following section, a short recap of this work is given. Further details can be found in [33].

Let $(k_1, r_1), \ldots, (k_n, r_n)$ be the sequence of pairs of keys and ranks for a dataset. The set of keys $k_1, \ldots, k_n$ imply a collection of subsequences in the key domain such that each subsequence is comprised of consecutive non-occupied keys. The sequence S is defined such that the element $S(i)$ corresponds to the i-th smallest key among all the endpoints from all subsequences.

The attacker first calculates the effect of inserting the first potential poisoning key $S(1)$, which implies the calculation of the values $M_K(1), M_{K^2}(1), M_{KR}(1)$, and $L(1)$. The effect of inserting poisoning key $S(i+1)$ on the loss function $L(i+1)$, can be computed in constant time via:

$$M_K(i+1) = M_K(i) + \frac{\Delta S(i)}{n+1}, \; M_{K^2}(i+1) = M_{K^2}(i) + \frac{(2S(i) + \Delta S(i))\,\Delta S(i)}{n+1}, \; M_R(i) = \frac{n+2}{2}, \; M_{R^2}(i) = \frac{(n+2)(2n+3)}{6}, \; M_{KR}(i+1) = M_{KR}(i) + \frac{T(i)\Delta S(i)}{(n+1)}$$

$$L(i+1) = -\frac{(M_{KR}(i+1) - M_K(i+1)M_R(i+1))^2}{M_{K^2}(i+1) - (M_K(i+1))^2} + M_{R^2}(i+1) - (M_R(i+1))^2$$

The above calculation maximizes the error of the poisoning for a single poisoning key in $O(n)$ time complexity. It runs as a subroutine of the GreedyDistributedPoisoningRegressionCDF algorithm, see line 3 - 10 in algorithm 1.

5.3.3 GreedyDistributedPoisoningRegressionCDF Algorithm

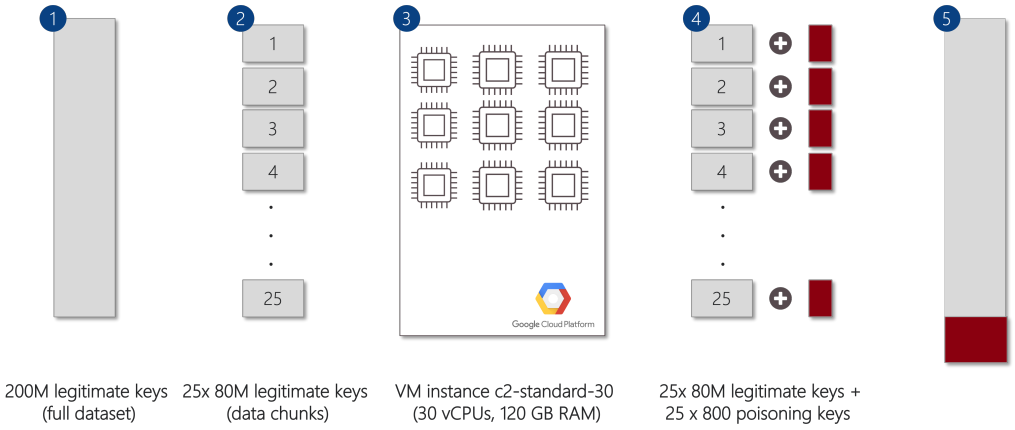


Figure 10: Procedure for distributed poisoning attack on large keysets.

As a contribution of this thesis, the GreedyPoisoningRegressionCDF algorithm described by [33] was extended to make it scale to very large datasets (> 200M keys). Specifically, the constraint on the greedy approach, where at each iteration the attacker makes a locally optimal decision and inserts the poisoning key that maximizes the error of the whole augmented keyset, has been relaxed. The result of this work is the *GreedyDistributedPoisoningRegressionCDF* algorithm which is presented in algorithm 1.

The core idea of *GreedyDistributedPoisoningRegressionCDF* is to distribute the dataset between multiple processing cores, with each core being responsible for poisoning only a part of the original dataset. Because executing the poisoning attack is computationally intensive, the author of this work has used a compute-optimized Virtual Machine instance of type `c2-standard-30` running on Google Compute Engine[1] with 30 vCPUs and 123 GB of RAM. Figure 10 shows the process of poisoning a 200M key dataset with 25 CPU cores.

[1]Google Cloud Platform - Compute Engine: `https://cloud.google.com/compute`

Algorithm 1: GREEDY DISTRIBUTED POISONING REGRESSION CDF

Data: The number of allowed poisoning keys p, the original dataset for the regression $\{(k_1, r_1), \ldots, (k_n, r_n)\}$ where $k_i \in K$ and $r_i \in [1, n]$, the number of CPUs m.
Result: Set of poisoning keys P such that $P \cap K = \emptyset$ and $|P| = p$.

Initialize the set of poisoning keys $P \leftarrow \emptyset$;
Split the original dataset into m chunks with size $\frac{n}{m}$;
Distribute the data chunks to the CPU cores along with number of poisoning keys $\frac{p}{n}$;
for *every* j *from 1 to* $\frac{p}{n}$ **do**
    Partition the non-occupied keys, i.e., keys not in $K \cup P$, into subsequences such that each subsequence consists of *consecutive* non-occupied keys;
    `//` `Due to convexity, the loss function is maximized at an endpoint`
    Extract the endpoints of each subsequence and sort them to construct the new sequence of endpoints $S(i)$, where $i \leq 2(n + j)$;
    Randomly sample the sequence of endpoints $S(i)$ without replacement to reduce the number of potential endpoints;
    Compute the rank that key $S(i)$ would have if it was inserted in $K \cup P$ and assign this rank as the i-th element of the new sequence $T(i)$, where $i \leq 2(n + j)$;
    `//` `Evaluate each sequence for the smallest endpoint`
    Compute the effect of choosing $S(1)$ as a poisoning key and inserting it to $K \cup P$ with the appropriate rank adjustments. Specifically, evaluate the sequences each of which is the mean M for a different variable, e.g., K, R, KR. Compute $M_K(1), M_{K^2}(1), M_{KR}(1)$, and $L(1)$;
    for *every* i *from 2 to the length of sequence* S **do**
        Compute the effect of choosing $S(i + 1)$ as a poisoning key by calculating the loss function $L(i + 1)$ from the equations in **??**;
    end
    Define as $k_{\text{OPT}} \leftarrow S(\arg\max_i L(i))$ the chosen poisoning key which maximizes the loss;
    Augment P as $P \leftarrow P \cup k_{\text{OPT}}$;
end
Collect the set of poisoning keys P from the CPUs;
return *the set of poisoning keys* P;

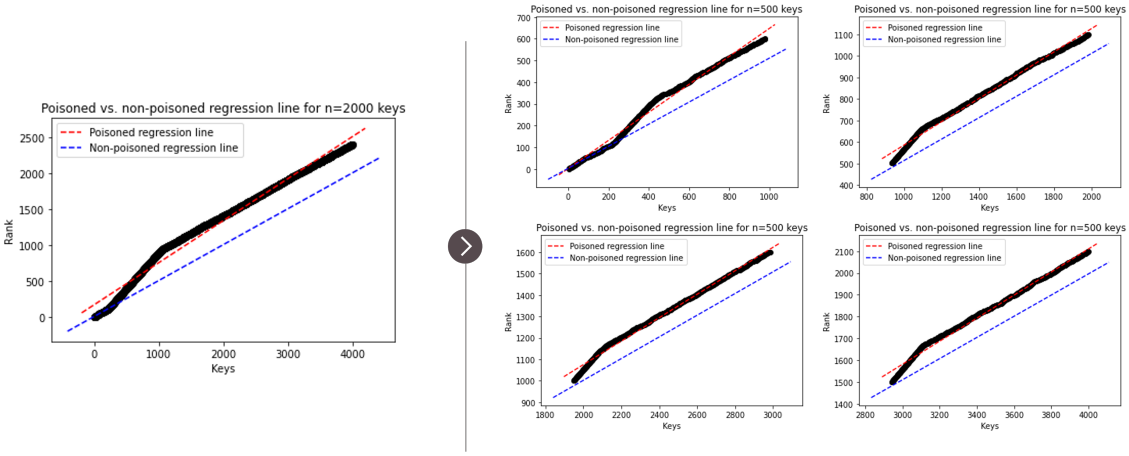


Figure 11: GreedyDistributedPoisoningRegressionCDF attack on dataset with n=2000 keys.

An illustration of the distributed poisoning attack is shown in Figure 11. It presents the application of the distributed poisoning algorithm using 4 CPU cores on a dataset of n=2000 keys. It can be observed that the greedy approach places poisoning keys in a dense area to exacerbate the non-linearity of the CDF and consequently increase the error.

Chapter

6

Evaluation

6.1 Experimental Setup

To evaluate the effectiveness of the poisoning attack described in Chapter 4, the author of this work has implemented a novel benchmarking system that facilitates the execution of multiple experiments with poisoned and non-poisoned datasets. The overall system architecture is shown in Figure 12 and further outlined below.

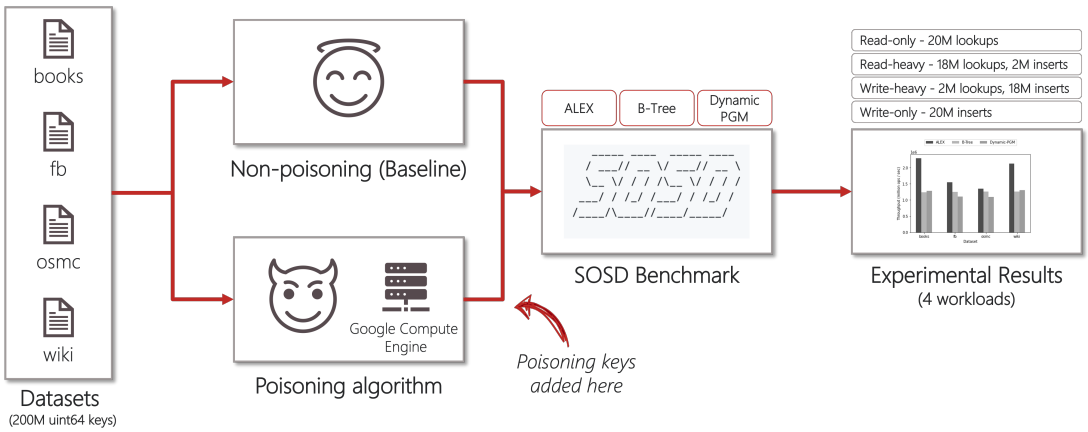


Figure 12: Benchmarking architecture used for experiments.

As shown in Figure 12, the evaluation is performed by using four real-world datasets that consist of 200 million keys each. These datasets are described in detail in Section 6.1.3 and are called *non-poisoned datasets*. To execute the poisoning attack, each of the datasets is poisoned by executing the attack described in Chapter 5 with a poisoning threshold of $p = 0.0001$. The result obtained from executing the poisoning attack is referred to as *poisoned datasets*.

In the next step, both the poisoned and non-poisoned datasets are evaluated across a variety of different workload scenarios as described in Section 6.1.2. To measure the impact of the poisoning attack in terms of deterioration of throughout of the LIS models, the SOSD benchmark published by Kipf et al. [37] is used. The source code for the SOSD benchmark is available online as open-source [42], but had to be significantly extended for the purpose of this research. The modifications to the source code are available online in this author's GitHub repository[2].

[2]Modified version of the SOSD Benchmark: `https://github.com/Bachfischer/SOSD/tree/develop`.

The main contributions of this research that were incorporated into the SOSD benchmark are:

- extension of SOSD benchmark to facilitate write operations
- extension of SOSD benchmark to support Dynamic PGM-Index
- modifications to measure index performance via throughput (instead of lookup latencies)
- update of the source code for indexes under evaluation to their latest version.

The evaluation of the learned indexes was performed on a Virtual Machine instance of type `e2-standard-8` running on Google Compute Engine with 8 vCPUs and 32 GB of RAM.

6.1.1 Evaluated Indexes

The experiments performed in this research focus on indexes that support dynamic updates (insert and delete operations) and whose source code is readily available to the community as open-source. These indexes are ALEX, Dynamic-PGM and B+ tree.
The RMI index [1] does not support write operations and was therefore excluded in the evaluation. The LIPP index supports write operations and is available to the research community as open-source code on Github [43]. However, after performing initial experiments for this research, the LIPP model was deemed to not be sufficiently reliable and scalable to very large datasets such as those used in this thesis. Due to these shortcomings, the LIPP index was excluded from the evaluation performed in this work.

Focusing on dynamic LIS models allows the comparison across different workloads (read- and writes), similar to what would be observed in a real-world deployment. Further details on the evaluated indexes as well as links to their respective source code repository are given in Table 3.

Method	Description	Type	Hyperparameters	Source Code
ALEX	[2]	Learned	-	[41]
Dynamic PGM-Index	[12]	Learned	maximum error ϵ	[44]
B+ tree	[45]	Tree	-	[46]

Table 3: Overview of evaluated indexes.

In the following section, a brief overview of the evaluated index implementations and their respective hyperparameters is given. All benchmarks were performed with multiple configurations (variants) of these indexes. To obtain the final results reported in this research, the overall average across all configurations was calculated.

Learned indexes:

The learned indexes used in this experiment are the ALEX [2] and PGM-Index [12] implementations (overview in Section 2.1).

The standard variant of the PGM-Index does not support dynamic updates, so for the purpose of this research, the dynamic version of the PGM-index (Dynamic PGM-Index) was used [44]. The PGM-Index provides a tunable hyperparameter ϵ that controls the maximum error bound during lookups. To evaluate the performance of the Dynamic PGM-Index under different configurations, multiple variations of the index were tested by varying the ϵ parameter during index creation.

The ALEX implementation supports dynamic updates by default and does not require the tuning of any hyperparameters [41]. Therefore, to evaluate the performance of ALEX under different configurations, the size of the index was varied by adjusting the number of keys that are inserted during index creation. This is done by inserting only every i-th key when initially constructing / bulk-loading the index from the datasets.

Trees:

A B+ tree [45] is a traditional in-memory index structures. Like ALEX, the B+ tree data structure does not require any hyperparameters to be tuned [46]. Instead, and similar to ALEX, during evaluation the size of the B+ tree was varied by adjusting the number of keys that are inserted during construction of the index.

6.1.2 Workloads

The primary metric for evaluating the performance of the learned indexes is the throughput in terms of million operations per second. The throughput is measured against a variety of workloads, each workload consisting of 20M keys:

(a) a read-only workload

(b) a read-heavy workload with 90% reads and 10% inserts,

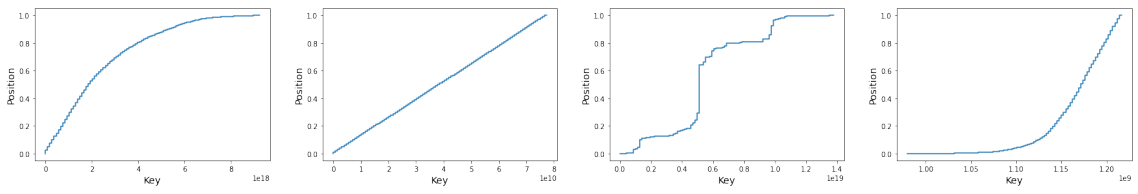
(c) a write-heavy workload with 10% reads and 90% inserts,

(d) a write-only workload, to complete the read-write spectrum.

First, for each workload, the index is constructed by bulk-loading it with 200M keys from the given dataset. Next, to measure read throughput of the index, a certain number of lookup keys is selected from the dataset and the index is queried for the keys' position. To measure write (insert) throughput, random keys that are not present in the dataset are generated and inserted into the index. For each operation, the time required for the execution of the workload is tracked and used to calculate the average throughput in million operations per second.

6.1.3 Datasets

Previous research has shown that learned indexes adapt well to synthetic data sampled from probability distributions [38]. To make the experimental setup of this research as realistic as possible, four real-world datasets from the SOSD benchmark [37] were used: *books*, *fb*, *osmc* and *wiki*. The datasets have already been used in previous experiments by the research community [39], [37] and are available online [47].



(a) CDF for books **(b)** CDF for fb **(c)** CDF for osmc **(d)** CDF for wiki

Figure 13: CDFs of evaluated datasets [47].

Each dataset from the SOSD benchmark consists of 200M 64-bit unsigned integer keys. The CDFs of the four datasets are shown in Figure 13, and detailed characteristics of the evaluated datasets are listed in Table 4.

	books	**fb**	**osmc**	**wiki**
Num keys:	200M	200M	200M	200M
Key type:	64-bit uint	64-bit uint	64-bit uint	64-bit uint
Total size:	1.53GB	1.53GB	1.53GB	1.53GB
Uniqueness:	Unique	Unique	Duplicates	Duplicates

Table 4: Dataset characteristics

books: This dataset is based on book popularity data from Amazon. Each key in the dataset represent the popularity of a particular book on Amazon.

fb: The keys in this dataset represent Facebook user IDs, where each key uniquely identifies a particular user. This dataset contains 21 outliers at the upper end of the key space that are several orders of magnitude larger than the rest of the keys.

osmc: The keys in this dataset represent cell IDs on OpenStreetMap. This dataset has clusters that are artifacts of projecting two-dimensional data into one-dimensional space [38].

wiki: The keys in this dataset represent edit timestamps on Wikipedia. Each key represents the time an edit was committed.

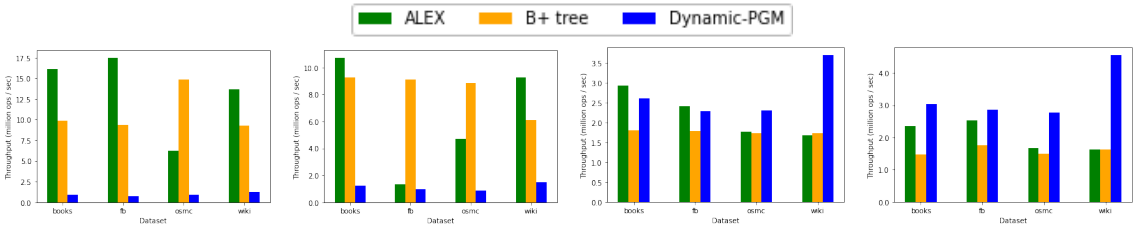
6.2 Experimental Results

The results of the evaluation are presented in Figure 14 and Figure 15 and detailed results are available in the Appendix. Figure 14 shows the performance of the learned index models in the non-poisoned setting, while Figure 15 shows the performance after the poisoning attack has been executed, i.e., after the insertion of 20.000 malicious poisoning keys.

From the plots, it can be seen that ALEX dominates B+ tree and Dynamic-PGM in almost all *Read-only* and *Read-heavy* workloads. In cases where the CDF is difficult to approximate (such as with the *osmc* dataset), B+ tree also performs well with approximately 9 million ops / second for the *Read-heavy workloads*. In the *Write-heavy* and *Write-only* workloads, Dynamic PGM dominates the other data structures across almost all datasets.

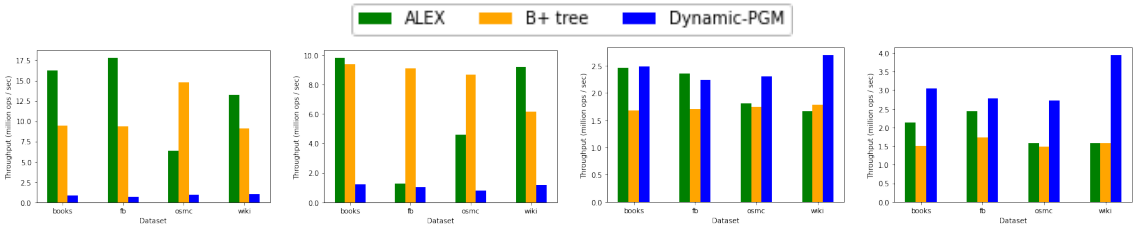
Figure 16 shows the performance deterioration in % between the non-poisoned and poisoned workload. As shown in the plots, introducing only a tiny subset of poisoning keys into the datasets (the experiments were performed with a poisoning threshold of $p = 0.0001$) can already cause a major performance deterioration in terms of throughput. Out of the evaluated data structures, Dynamic-PGM seems to be particularly prone to poisoning attacks, especially when evaluated on the *wiki* dataset.

The performance of ALEX also fluctuates and deteriorates significantly after the poisoning attack is executed on the *books* dataset. The "negative" performance deterioration on the *Read-only*



(a) Read-only workload **(b)** Read-heavy workload **(c)** Write-heavy workload **(d)** Write-only workload

Figure 14: Comparison of throughput for different LIS models (Non-poisoned scenario).

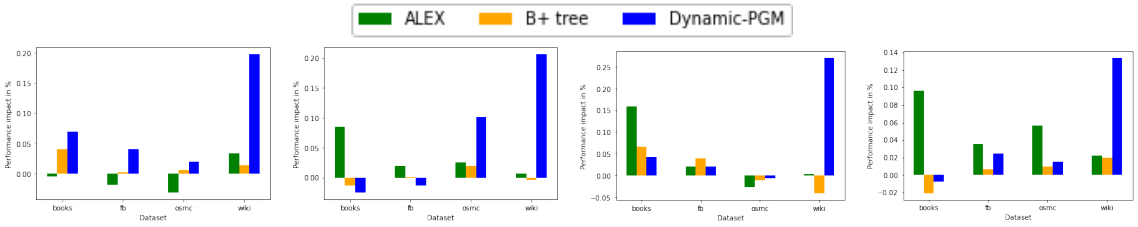


(a) Read-only workload **(b)** Read-heavy workload **(c)** Write-heavy workload **(d)** Write-only workload

Figure 15: Comparison of throughput for different LIS models (Poisoned scenario).

workload for the *fb* and *osmc* datasets indicate a minor speed-up of the ALEX model after it is being evaluated on the poisoned dataset. However, the difference between the poisoned and non-poisoned scenario is only minor and therefore negligible.

Another interesting observation is that the performance of the B+ tree data structure does not seem to be impacted by the poisoning attack, as it remains stable with an performance impact of less than 5% across all workloads and datasets. This can attributed to the tree-like structure of a B+ tree which exhibits proven worst-case guarantees in terms of algorithmic complexity.



(a) Read-only workload **(b)** Read-heavy workload **(c)** Write-heavy workload **(d)** Write-only workload

Figure 16: Performance deterioration of LIS models under adversarial workload.

The experimental evaluation shows that LIS models are prone to poisoning attacks and lose their perceived performance benefits over traditional data structures like B+ trees when they are tested on datasets with a CDF that is difficult to approximate. This gives rise to the remark that “adversarial workload matters” because it forces the data structure to exhibit its worst-case algorithmic complexity.

Chapter

7

Discussion

7.1 Limitations

Even though the experimental results shown in this thesis are highly convincing, a variety of limitations apply.

The poisoning attack described in this research was executed with a poisoning threshold of $p = 0.0001$, which yields 20.000 poisoning keys for a dataset containing 200M keys in total. Contrary to the large-scale attack executed in this thesis, previous work on poisoning attacks has targeted much smaller key sizes (only up to 10.000 keys), but has made use of a larger poisoning threshold (typically in the range of $p = 0.01$ to $p = 0.2$). Because of the massive amount of computational resources required to generate a larger poisoning key set, the author of this work has abstained from performing experiments with larger poisoning thresholds. It is assumed that executing the poisoning attack with a larger poisoning threshold would increase the effects described in Section 6.2 even further.

As described in Chapter 6, the experiments in this research were performed on a *Virtual Machine* (VM) instance of type `e2-standard-8` running on Google Compute Engine with 8 vCPUs and 32 GB of RAM. Because the VM instance is running on a shared physical host along other VMs, there is the potential that the VM could exhibit slightly different performance characteristics depending on the I/O demand caused by the other VM instances running in parallel on the same host.

7.2 Recommendations

This work has shown how a large-scale poisoning attack against dynamic LIS models can be executed. Potential mitigation of poisoning attacks have been studied extensively in recent years, but were focused primarily on neural network models [48] [49], [50]. Approaches for defending against poisoning attacks on linear regression models are less prevalent in the literature.

In [31], Jagielski et al. proposed a poisoning detection algorithm for linear regressions called TRIM. It recovers the legitimate non-poisoned dataset by searching for the keys that cause the largest loss and identify them as poisoning keys. Another potential mechanism for defending regression models against poisoning attacks has recently been proposed by Weerasinghe et al. [51] and is based on measuring the deviation of a given data point's local *Local Intrinsic Dimensionality* (LID) with respect to its neighbors. Through experimental evaluation, Weerasinghe et al. were able to show the effectiveness of the defense mechanism and consistently outperformed other defenses such as TRIM.

Two major limitations arise when applying TRIM or the LID algorithm to defend against the poisoning attack described in this work. Firstly, in the LIS setting, the rank for each key depends on the value of all other keys in the dataset; this implies that the mitigation has to iteratively re-calibrate its parameters and as a result become extremely inefficient. Secondly, the poisoning keys are typically concentrated around legitimate keys. It is therefore assumed that no known mitigation can exclusively remove poisoning keys without also removing a significant amount of legitimate keys.

7.3 Future Directions

This research has successfully executed a large-scale poisoning attack on dynamic learned index models by poisoning the CDF of the data that they try to approximate. This work was focused exclusively on poisoning models that try to approximate the CDF via linear regression. Recent publications have introduced other models for constructing a LIS, such as polynomial interpolation [4] and logarithmic error regression [40]. These models also provide an interesting target for poisoning attacks.

While the poisoning attack described here works by introducing poisoning keys prior to training the index models, future research may also choose to investigate how an adversary could leverage the update functionality of dynamic learned index models to insert and remove keys from a trained LIS model at runtime to deteriorate the fit of the LIS.

Chapter

8

Conclusion

The recent emergence of LIS has been driven by the desire to overhaul the architecture of modern database systems by replacing traditional data structures such as B+ trees with machine learning models. This is based on the assumption that by replacing a B+ tree data structure with a ML model, a performance improvement from $O(\log n)$ to $O(1)$ is theoretically possible.

Using ML models to approximate the index of a database works by training a LIS model on the available data, i.e., keyset. In this work, a poisoning attack for very large keysets (up to 200M keys) was executed. It builds on the observation that LIS models try to approximate the CDF of the keyset by training series of linear regression models. By executing the poisoning attack, a malicious adversary poisons the keyset prior to constructing the LIS.

The poisoning attack described in this thesis was mounted against two dynamic LIS models and one traditional data structure: ALEX, Dynamic-PGM and B+ tree. The evaluation was performed using four different datasets that are part of the SOSD benchmark [37]. To evaluate the success of the poisoning attack, the throughput of the indexes against a series of workload scenarios was measured. The experimental evaluation of the LIS models has shown that learned index structures like ALEX and Dynamic PGM-Index are prone to poisoning attacks and exhibit significantly worse performance after only a small subset of poisoning keys is introduced into the keyset (poisoning threshold $p = 0.0001$). In contrast, the B+ tree data structure is relatively robust and exhibits no significant performance deterioration under adversarial workload.

The results from this research motivate the study of worst-case performance behavior of LIS models. The poisoning attack and benchmark harness described in this work lay the foundation for future study of LIS under an adversarial setting and can be easily adapted for future research purposes.

List of Figures

List of Tables

Acronyms

ALEX	Adaptive Learned indEX
CDF	Cumulative Distribution Function
COLIN	Cache-cOnscious Learned INdex
DAG	Directed Acyclic Graph
FMCD	Fastest Minimum Conflict Degree
GA	Gapped Array
LID	Local Intrinsic Dimensionality
LIPP	Learned Index with Precise Positions
LIS	Learned Index Structure
LS	Linear Spline
MSE	Mean Squared Error
ML	Machine Learning
PGM	Piecewise Geometric Model
PLA	Piecewise Linear Approximation
RMI	Recursive Model Index
RS	Radix Spline
VM	Virtual Machine

Bibliography

[1] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis: *The case for learned index structures.* In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504, 2018.

[2] Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, and Donald Kossmann: *Alex: an updatable adaptive learned index.* In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 969–984, 2020.

[3] Yifan Dai, Yien Xu, Aishwarya Ganesan, Ramnatthan Alagappan, Brian Kroth, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau: *From wisckey to bourbon: A learned index for log-structured merge trees.* In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 155–171, 2020, ISBN 193913319X.

[4] Naufal Fikri Setiawan, Benjamin IP Rubinstein, and Renata Borovica-Gajic: *Function interpolation for learned index structures.* In *Australasian Database Conference*, pages 68–80. Springer, 2020.

[5] Ali Hadian and Thomas Heinis: *Interpolation-friendly b-trees: Bridging the gap between algorithmic and learned indexes.* 2019.

[6] Hussam Abu-Libdeh, Deniz Altınbüken, Alex Beutel, Ed H Chi, Lyric Doshi, Tim Kraska, Andy Ly, and Christopher Olston: *Learned indexes for a google-scale disk-based database.* arXiv preprint arXiv:2012.12501, 2020.

[7] Chuzhe Tang, Zhiyuan Dong, Minjie Wang, Zhaoguo Wang, and Haibo Chen: *Learned indexes for dynamic workloads.* arXiv preprint arXiv:1902.00655, 2019.

[8] Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra: *A "learned" approach to quicken and compress rank/select dictionaries.* In *2021 Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 46–59. SIAM, 2021.

[9] Vikram Nathan, Jialin Ding, Mohammad Alizadeh, and Tim Kraska: *Learning multi-dimensional indexes.* In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 985–1000, 2020.

[10] Stephen Macke, Alex Beutel, Tim Kraska, Maheswaran Sathiamoorthy, Derek Zhiyuan Cheng, and EH Chi: *Lifting the curse of multidimensional data with learned existence indexes.* In *Workshop on ML for Systems at NeurIPS*, 2018.

[11] Darryl Ho, Jialin Ding, Sanchit Misra, Nesime Tatbul, Vikram Nathan, Vasimuddin Md, and Tim Kraska: *Lisa: towards learned dna sequence search.* arXiv preprint arXiv:1910.04728, 2019.

[12] Paolo Ferragina and Giorgio Vinciguerra: *The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds.* Proceedings of the VLDB Endowment, 13(10):1162–1175, 2020, ISSN 2150-8097.

[13] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann: *Radixspline: a single-pass learned index.* In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–5, 2020.

[14] Jialin Ding, Vikram Nathan, Mohammad Alizadeh, and Tim Kraska: *Tsunami: A learned multi-dimensional index for correlated data and skewed workloads.* arXiv preprint arXiv:2006.13282, 2020.

[15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska: *Fiting-tree: A data-aware index structure*, 2019. https://doi.org/10.1145/3299869.3319860.

[16] Michael Mitzenmacher: *A model for learned bloom filters, and optimizing by sandwiching.* arXiv preprint arXiv:1901.00902, 2019.

[17] Ali Hadian and Thomas Heinis: *Considerations for handling updates in learned index structures.* In *Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, pages 1–4, 2019.

[18] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing: *Updatable learned index with precise positions.* arXiv preprint arXiv:2104.05520, 2021.

[19] Zhou Zhang, Pei Quan Jin, Xiao Liang Wang, Yan Qi Lv, Shou Hong Wan, and Xi Ke Xie: *Colin: A cache-conscious dynamic learned index with high read/write performance.* Journal of Computer Science and Technology, 36(4):721–740, 2021, ISSN 1860-4749.

[20] Mark H Overmars: *The design of dynamic data structures*, volume 156. Springer Science and Business Media, 1987, ISBN 354012330X.

[21] Ryan Marcus, Emily Zhang, and Tim Kraska: *Cdfshop: Exploring and optimizing learned index structures.* In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 2789–2792, 2020.

[22] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and J Doug Tygar: *Adversarial machine learning.* In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58, 2011.

[23] Marco Barreno, Blaine Nelson, Anthony D Joseph, and J Doug Tygar: *The security of machine learning.* Machine Learning, 81(2):121–148, 2010, ISSN 1573-0565.

[24] Micah Goldblum, Dimitris Tsipras, Chulin Xie, Xinyun Chen, Avi Schwarzschild, Dawn Song, Aleksander Madry, Bo Li, and Tom Goldstein: *Data security for machine learning: Data poisoning, backdoor attacks, and defenses.* arXiv preprint arXiv:2012.10544, 2020.

[25] Luis Muñoz-González, Battista Biggio, Ambra Demontis, Andrea Paudice, Vasin Wongrassamee, Emil C Lupu, and Fabio Roli: *Towards poisoning of deep learning algorithms with back-gradient optimization.* In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 27–38, 2017.

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

[27] Sanli Tang, Xiaolin Huang, Mingjian Chen, Chengjin Sun, and Jie Yang: *Adversarial attack type i: Cheat classifiers by significant changes.* IEEE transactions on pattern analysis and machine intelligence, 2019, ISSN 0162-8828.

[28] Battista Biggio and Fabio Roli: *Wild patterns: Ten years after the rise of adversarial machine learning.* Pattern Recognition, 84:317–331, 2018, ISSN 0031-3203.

[29] Chang Liu, Bo Li, Yevgeniy Vorobeychik, and Alina Oprea: *Robust linear regression against training data poisoning.* In *Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security*, pages 91–102, 2017.

[30] Huang Xiao, Battista Biggio, Gavin Brown, Giorgio Fumera, Claudia Eckert, and Fabio Roli: *Is feature selection secure against training data poisoning?* In *international conference on machine learning*, pages 1689–1698. PMLR, 2015.

[31] Matthew Jagielski, Alina Oprea, Battista Biggio, Chang Liu, Cristina Nita-Rotaru, and Bo Li: *Manipulating machine learning: Poisoning attacks and countermeasures for regression learning.* In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 19–35. IEEE, 2018, ISBN 1538643537.

[32] Nicolas Müller, Daniel Kowatsch, and Konstantin Böttinger: *Data poisoning attacks on regression learning and corresponding defenses.* In *2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC)*, pages 80–89. IEEE, 2020, ISBN 1728180031.

[33] Evgenios M Kornaropoulos, Silei Ren, and Roberto Tamassia: *The price of tailoring the index to your data: Poisoning attacks on learned index structures.* arXiv preprint arXiv:2008.00297, 2020.

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

[35] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli: *Evasion attacks against machine learning at test time.* In *Joint European conference on machine learning and knowledge discovery in databases*, pages 387–402. Springer, 2013.

[36] Shike Mei and Xiaojin Zhu: *Using machine teaching to identify optimal training-set attacks on machine learners.* In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[37] Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann: *Sosd: A benchmark for learned indexes.* arXiv preprint arXiv:1911.13014, 2019.

[38] Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska: *Benchmarking learned indexes.* arXiv preprint arXiv:2006.12804, 2020.

[39] Marcel Maltry and Jens Dittrich: *A critical analysis of recursive model indexes.* arXiv preprint arXiv:2106.16166, 2021.

[40] Martin Eppert, Philipp Fent, and Thomas Neumann: *A tailored regression for learned indexes: Logarithmic error regression.* 2021.

[41] Microsoft: *Alex - a library for building an in-memory, adaptive learned index*, 2021. https://github.com/microsoft/ALEX, visited on 2021-07-08.

[42] *Sosd benchmark*, 2021. https://github.com/learnedsystems/SOSD, visited on 2021-07-08.

[43] Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing: *Lipp: Updatable learned index with precise positions*, 2021. https://github.com/Jiacheng-WU/lipp/, visited on 2021-07-08.

[44] Paolo Ferragina and Giorgio Vinciguerra: *Pgm-index: State-of-the-art learned data structure*, 2021. https://github.com/gvinciguerra/PGM-index, visited on 2021-07-08.

[45] R. Bayer and E. McCreight: *Organization and maintenance of large ordered indices.* https://doi.org/10.1145/1734663.1734671.

[46] *Stx b+ tree*, 2021. https://panthema.net/2007/stx-btree/, visited on 2021-07-08.

[47] Ryan Marcus, Andreas Kipf, and Alex van Renen: *Searching on Sorted Data*, 2019. https://doi.org/10.7910/DVN/JGVF9A.

[48] Jayaram Raghuram, Varun Chandrasekaran, Somesh Jha, and Suman Banerjee: *A general framework for detecting anomalous inputs to dnn classifiers*, 2021. http://proceedings.mlr.press/v139/raghuram21a.html.

[49] Ahmed Salem, Yang Zhang, Mathias Humbert, Pascal Berrang, Mario Fritz, and Michael Backes: *Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models.* arXiv preprint arXiv:1806.01246, 2018.

[50] Bolun Wang, Yuanshun Yao, Shawn Shan, Huiying Li, Bimal Viswanath, Haitao Zheng, and Ben Y Zhao: *Neural cleanse: Identifying and mitigating backdoor attacks in neural networks.* In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 707–723. IEEE, 2019, ISBN 153866660X.

[51] Sandamal Weerasinghe, Sarah M Erfani, Tansu Alpcan, Christopher Leckie, and Justin Kopacz: *Defending regression learners against poisoning attacks.* arXiv preprint arXiv:2008.09279, 2020.

Chapter

A

Experimental Results

This chapter shows the results from the evaluation of learned indexes under different workloads using the non-poisoned and poisoned datasets. The workloads correspond to:

(a) a read-only workload

(b) a read-heavy workload with 90% reads and 10% inserts,

(c) a write-heavy workload with 10% reads and 90% inserts,

(d) a write-only workload, to complete the read-write spectrum.

A.1 Non-poisoned scenario

	ALEX	B+ tree	Dynamic PGM Index
books	16162215	9863725	926508
fb	17498954	9399892	731971
osmc	6194756	14897510	917548
wiki	13687183	9259387	1268112

Workload (a): Throughput of evaluated LIS models in ops / second (20M lookups).

	ALEX	B+ tree	Dynamic PGM Index
books	10728106	9238867	1215244
fb	1313206	9088619	998036
osmc	4704530	8839391	872240
wiki	9263433	6119975	1473184

Workload (b): Throughput of evaluated LIS models in ops / second (18M lookups, 2M inserts).

	ALEX	B+ tree	Dynamic PGM Index
books	2921892	1796295	2605189
fb	2408246	1774400	2289189
osmc	1770611	1732262	2295980
wiki	1676874	1720761	3698999

Workload (c): Throughput of evaluated LIS models in ops / second (2M lookups, 18M inserts).

	ALEX	B+ tree	Dynamic PGM Index
books	2353992	1477951	3028375
fb	2534546	1749967	2846518
osmc	1677064	1495389	2761089
wiki	1615591	1615169	4558578

Workload (d): Throughput of evaluated LIS models in ops / second (20M inserts).

A.2 Poisoned scenario

	ALEX	B+ tree	Dynamic PGM Index
books	16243040	9460745	861921
fb	17821675	9384162	702158
osmc	6387984	14809695	899915
wiki	13233802	9134914	1017923

Workload (a): Throughput of evaluated LIS models in ops / second (20M lookups).

	ALEX	B+ tree	Dynamic PGM Index
books	9819021	9366380	1245485
fb	1286960	9083855	1011879
osmc	4586141	8668881	784075
wiki	9204752	6143775	1169301

Workload (b): Throughput of evaluated LIS models in ops / second (18M lookups, 2M inserts).

	ALEX	B+ tree	Dynamic PGM Index
books	2457741	1677875	2494472
fb	2358933	1705110	2240320
osmc	1816547	1751328	2310215
wiki	1672190	1790405	2701049

Workload (c): Throughput of evaluated LIS models in ops / second (2M inserts, 18M lookups).

	ALEX	B+ tree	Dynamic PGM Index
books	2128055	1508457	3050597
fb	2445033	1738097	2777453
osmc	1581750	1481324	2719388
wiki	1579460	1582920	3949376

Workload (d): Throughput of evaluated LIS models in ops / second (20M inserts).