A Dynamic Piecewise-Linear Geometric Index with Worst-Case Guarantees

Emil Toftegaard Gæde
Technical University of Denmark, Lyngby, Denmark

Ivor van der Hoog
IT University of Copenhagen, Denmark

Eva Rotenberg
IT University of Copenhagen, Denmark

Tord Stordalen
Technical University of Denmark, Lyngby, Denmark

Abstract

Indexing data is a fundamental problem in computer science. The input is a set S of n distinct integers from a universe $\mathcal{U}$. Indexing queries take a value $q \in \mathcal{U}$ and return the `membership`, `predecessor` or `rank` of q in S. A `range` query takes two values $q, r \in \mathcal{U}$ and returns the set $S \cap [q, r]$.

Recently, various papers study a special case where the the input data behaves in an approximately piece-wise linear way. Given the sorted (rank,value) pairs, and given some constant ε, one wants to maintain a small number of axis-disjoint line-segments such that, for each rank, the value is within $\pm\varepsilon$ of the corresponding line-segment. Ferragina and Vinciguerra (VLDB 2020) observe that this geometric problem is useful for solving indexing problems, particularly when the number of line-segments is small compared to the size of the dataset.

We study the dynamic version of this geometric problem. In the dynamic setting, inserting or deleting just one data point may cause up to three line-segments to be merged, or one line-segment to be split at most three-way. To determine and compute this, we use techniques from dynamic maintenance of convex hulls, and provide new algorithms with worst-case guarantees, including an $O(\log n)$ algorithm to compute a separating line between two non-intersecting convex hulls – an operation previously missing from the literature.

We then use our fully-dynamic geometry-based subroutine in an indexing data structure, combining it with a natural hashing technique. The resulting indexing data structure has theoretically efficient worst-case guarantees in expectation. We compare its practical performance to the solution of Ferragina and Vinciguerra, which was shown to perform better in certain structured settings [Sun, Zhou, Li VLDB 2023]. Our empirical analysis shows that our solution supports more efficient range queries in the special case where the update sequence contains many deletions.

2012 ACM Subject Classification Theory of computation → Computational geometry

Keywords and phrases Algorithms Engineering, Data Structures, Indexing, Convex Hulls

Digital Object Identifier 10.4230/LIPIcs.ESA.2025.64

Related Version *Full Version*: `https://arxiv.org/abs/2503.05007`

Supplementary Material
Software (Source Code): `https://github.com/Sgelet/DynamicLearnedIndex` [18]
archived at `swh:1:dir:b8763eb0504d33beb81ee89d230a30dca8ab0b66`
Software (Test Bed): `https://github.com/Sgelet/LearnedIndexBench` [19]
archived at `swh:1:dir:07ea25cfc176438933c1a5507bfdad3ba9461ab6`

Funding This work was supported by the Carlsberg Foundation Fellowship CF21-0302 "Graph Algorithms with Geometric Applications", the VILLUM Foundation grant (VIL37507) "Efficient Recomputations for Changeful Problems", and the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 899987.

Acknowledgements We thank Linda Kleist for reading this paper and correcting our constants.

1 Introduction

We investigate the use of *learned indices* for the design of *dynamic indexing data structures*.

Indexing data structures. An indexing data structure maintains a set S of n distinct integers from a universe $\mathcal{U}$. Let RANK : $S \to [n]$ be the function mapping each $s \in S$ to its index in the sorted order of S. The objective is to support the following indexing queries:

- `member`(q) returns **`true`** if $q \in S$.
- `predecessor`(q) returns $\max\{t \in S \mid t < q\}$. (We allow $q \notin S$.)
- `rank`(q) returns RANK(`predecessor`(q)) $+ 1$. (We allow $q \notin S$.)

Additionally, we consider *range queries*, where k denotes the output size:

- `range`$(q,\ t)$ returns $S \cap [q, t]$. (We allow $q, t \notin S$.)

Static indexing data structures fall into three broad categories: *Tree-based solutions* store S in a sorted array A, requiring no additional space but incurring logarithmic query costs. Tree traversals enable `predecessor`, `rank`, and `member` queries in $O(\log n)$ time and range queries in $O(\log n + k)$ time [1, 3, 32, 37]. *Map-based solutions* store S in sorted order and maintain a hash map $H : S \to [n]$ mapping each element to its rank [4, 23, 31]. This enables constant-time support for `member`, `predecessor`, and `rank` queries if queries are restricted to elements of S, and $O(k)$ time for `range` queries when both endpoints lie in S. The additional space is $O(n)$. A third category use what are called *learned indices*, a recently introduced term [24, 20, 16, 22, 8]. Given an integer parameter ε, a learned index is a function $h_\varepsilon : \mathcal{U} \to [0, n]$ such that

$$h_\varepsilon(q) \in [\texttt{rank}(q) - \varepsilon, \texttt{rank}(q) + \varepsilon].$$

The function h_ε is learned from S and used to guide search in a sorted array A storing S. Ferragina and Vinciguerra [16] interpret h_ε geometrically: each $s \in S$ maps to a point $(\text{RANK}(s), s)$ in the plane, and h_ε is learned as a piecewise-linear approximation to F_S.

A notable instance of learned indices is the *PGM index* [16], where h_ε is a y-monotone piecewise-linear function made of segments, with the property that each point in F_S lies within an ε-wide horizontal strip around some segment. Let $|h_\varepsilon|$ denote the number of segments. The data structure supports indexing queries in $O(\varepsilon + \log |h_\varepsilon|)$ time and range queries in $O(\varepsilon + k + \log |h_\varepsilon|)$ time. They also show how to construct a PGM index in linear time, such that there exists no PGM index h'_ε with $|h_\varepsilon| > 2|h'_\varepsilon|$. Ferragina and Vinciguerra argue that learned indices are the "best of both worlds" since:

- the supported queries are as general as those supported by tree-based solutions,
- the solution uses only $O(|h_\varepsilon|)$ additional space, and
- $O(\varepsilon + \log |h_\varepsilon|)$ is, for an appropriate choice of ε, efficient in practice.

Their performance has been empirically benchmarked in several studies [15, 24, 38, 37].

Dynamic indexing data structures. Due to their fundamental role, dynamic indexing structures have received extensive theoretical and practical attention. When S is dynamic, maintaining a sorted array becomes inefficient. Tree-based structures can be updated in $O(\log n)$ time with tree rotations. Map-based approaches allow constant-time `member` updates but are typically not extended to support other indexing queries. Learned indices offer a promising direction by exploiting structural properties of S, akin to parametrised algorithms. However, just as parametrised algorithms, data structures based on learned indices are not always efficient: if S lacks exploitable structure or access patterns are skewed, traditional

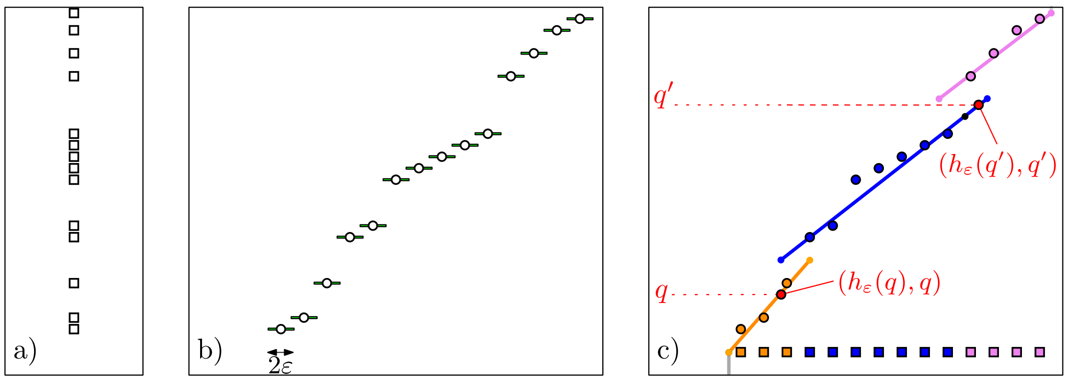

Figure 1 (a) a set of n values S. (b) S corresponds to an xy-monotone point set F_S. (c) The PGM index computes a y-monotone set of segments that starts and ends with a vertical halfline.

indexing data structures are preferred [33, 26]. Experimental studies have examined properties of learned indices [14, 13, 26, 33], in an effort to classify when they are appropriate to use. As a brief summary, learned indices pay a price in updates [26], and traditional indices are preferred if S or the access pattern is complex or skewed, or if concurrency is possible [33]. If there is sufficient structure, both space usage and access times can benefit from learned indices [13, 14], subject to the strategy employed by the learned index.

Dynamic learned indices through the logarithmic method. The logarithmic method of Overmars [29] provides an amortised way to maintain learned indices. S is partitioned into $\lceil \log n \rceil$ buckets B_i, each of size 2^i. Each bucket is either full or empty, and stores its contents in an array A_i in sorted order and maintains a learned index h^i_ε over A_i.

Let the learned index h_ε have a construction time of $T(n)$. This data structure can be maintained insertion-only in amortised $O(T(n) \log n)$ time. An insertion inserts a new value into B_0. Let j be the maximum integer such that all B_i for $i \in [0, j-1]$ are full. This approach empties these buckets, fills B_j in sorted order, and constructs (A_j, h^j_ε) in $O(T(2^j))$ time. Whenever we delete some $s \in S$, this approach instead inserts a *tombstone* s^*, which is a special copy of s. If an insertion fills a new bucket B_j, it first iterates over all elements. If B_j contains both s and s^*, it removes both elements. It then constructs (A_j, h^j_ε) twice. Once on all "normal" values, and once on all tombstones in B_j. This way, deletions take the same time as insertions do. In this paper, we consider the following open question:

"Can a learned index be dynamically be maintained with worst-case guarantees?"

Intermezzo: computing a line cover. The interpretation of learned indices by Ferragina and Vinciguerra [16] translates to a geometric problem where the goal is to (approximately) cover a monotone set of two-dimensional points by a set of line segments. Using the logarithmic method, and the static algorithm of O'Rourke [27] to approximately points, they dynamically maintain an *ε-cover*: a set of lines that are guaranteed to be within an ε horizontal distance from each point. We consider the problem of maintaining dynamic ε-covers to be an interesting geometric problem in its own right.

From learned indices to indexing data structures. Under the logarithmic method, indexing queries decompose naturally across the buckets:

- For `member`(q), $q \in S$ if and only if there exists an $i \in [\lceil \log n \rceil]$ with $q \in B_i$.
- For `predecessor`(q), the output is the maximum predecessor across B_i for $i \in [\lceil \log n \rceil]$.

- For $\texttt{rank}(q)$, the rank is the sum of all ranks of q in B_i for $i \in [\lceil \log n \rceil]$.
- For $\texttt{range}(q, t)$, the reported range is the union of all ranges in B_i for $i \in [\lceil \log n \rceil]$.

This way, indexing queries require only an additional factor $O(\log n)$ time. Indexing queries can be answered by combining queries to both the normal and tombstone structures. E.g., $\texttt{rank}(q)$ is the rank of q in the "normal" data structure minus the rank of q in the tombstone structure. This approach has two downsides:

- First, approach has an amortised update time.
- Second, this approach does not support output-sensitive range queries – as there may be $O(n)$ values s that (together with their tombstones s^*) lie in between a query pair (q, t).

This leads to the following open question:

> "Can a dynamic learned index be converted into an output-sensitive dynamic indexing data structure with worst-case guarantees?"

Contribution and organization. We propose maintaining a dynamic ε-cover (and thereby a dynamic learned index) via dynamic convex hull techniques. Section 3 shows that deciding whether S admits an ε-cover of complexity 1 is equivalent to convex hull intersection testing (Figure 2). Section 4 shows a robust algorithm to compute the intersection between two convex hulls in $O(\log n)$ time. We adapt our algorithm to output a separating line (which is a learned index of complexity 1) in the negative case. Section 5 combines these with dynamic convex hull data structures to yield a dynamic learned index worst-case $O(\log^2 n)$ update time. We empirically compare our learned index to the PGM index from [16].

Section 6 introduces a novel hashing-based approach to convert learned indices into dynamic indexing data structures, using $O(\varepsilon^{-1})$ additional expected overhead. We compare our dynamic indexing structure to the amortised PGM index of [16] in terms of update time and index complexity. We do not benchmark against traditional indexing data structures – since the relation between learned and traditional indices is previously studied [33]. Instead, our goal is to push the theoretical limits and worst-case guarantees of learned indices.

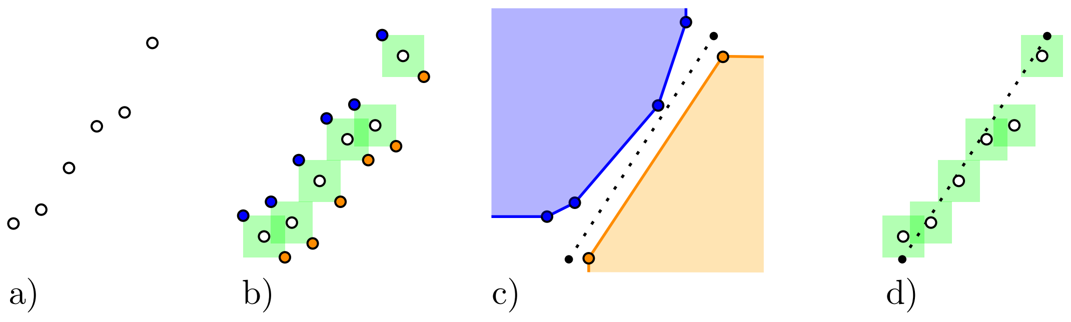

Figure 2 For any set F_S, we construct two convex hulls. We prove that there exists a segment ℓ within L_∞-distance ε of all points in F_S if and only if these hulls do not intersect. We adapt the convex hull intersection testing algorithm to find ℓ whenever these hulls are disjoint.

2 Preliminaries

The input is a dynamic set S of n distinct positive integers from some universe $\mathcal{U}$. For $a, b \in \mathbb{Z}$ with $a \leq b$, we define $S[a, b]$ as the set $S \cap [a, b]$. We denote by F_S the two-dimensional point set obtained by mapping each $s \in S$ to $(\text{RANK}(S), s)$. Throughout this paper, we distinguish between positions and strict positions. E.g., lying above or strictly above a line.

▶ **Definition 1** ([16]). *Let ε be a positive integer. A* PGM index *h_ε of S is defined as a y-monotone set of segments that together cover the y-axis. We regard h_ε as a map from y-coordinates to x-coordinates and require that for all $q \in \mathcal{U}$, $h_\varepsilon(q) \in [\texttt{rank}(q)-\varepsilon, \texttt{rank}(q)+\varepsilon]$.*

Ferragina and Vinciguerra [16, Lemma 1] wrongfully claim an $O(n)$-time algorithm to compute a minimum complexity PGM index h_ε. They invoke a streaming algorithm by O'Rourke [27] for fitting straight lines through data ranges. We show that this algorithm outputs a PGM index h_ε such that there exists no PGM index h'_ε with $|h_\varepsilon| > 2h'_\varepsilon$ (see the full version for details). Their algorithm restricts S to contain no duplicates. We assume the same setting and compute something slightly different as we define an ε-cover instead:

▶ **Definition 2.** *Let ε be a positive integer. We define an ε-cover f of S as a set of vertically separated segments with slope at least 1 where all $(r, s) \in F_S$ are within L_∞-distance ε of f.*

An ε-cover has a functionality and complexity similar to a learned index:

▶ **Observation 3.** *Let f be an ε-cover and Q be a horizontal line with height $q \in [\min S, \max S]$. Let (s, t) be the segment in f closest to q. Then $(line(s,t) \cap Q).x \in [\texttt{rank}(q)-2\varepsilon, \texttt{rank}(q)+2\varepsilon]$.*

▶ **Observation 4.** *For fixed ε, let k denote the minimum complexity of any PGM index of S. If f is an ε-cover of S of minimum complexity, then f contains at most $k-2$ edges.*

▶ **Definition 5.** *For any fixed ε-cover f of S, we define $\Lambda(f)$ as the set of pairwise interior-disjoint one-dimensional intervals that correspond to the y-coordinates of segments in f.*

Dynamic convex hulls. We dynamically maintain an ε-cover f of S of approximately minimum complexity. To this end, we use a result by Overmars and van Leeuwen [30] to dynamically maintain for all $[a, b] \in \Lambda(f)$ the convex hull of $F_{S[a,b]}$. For any point set F, denote by $CH(F)$ their convex hull. The data structure in [30] is a balanced binary tree over F, which at its root maintains a balanced binary tree over the edges $CH(F)$ in their cyclical ordering. It uses $O(n)$ space and has worst-case $O(\log^2 n)$ update time.

Rank-based convex hulls. For any update in S, up to n values in F_S may change their x-coordinate. This complicates the maintenance of a dynamic data structure over F. Gæde, Gørtz, van Der Hoog, Krogh, and Rotenberg [17] observe that all algorithmic logic in [30] requires only the *relative* x-coordinates between points. They adapt [30] to give an efficient and robust implementation of what they call *a rank-based convex hull* data structure $T(S)$ with $O(\log^2 n)$ update time. For ease of exposition, we overly simplify their functionality:

For each $[a, b] \in \Lambda(f)$, we store $S[a, b]$ in $T(S[a, b])$. $T(S[a, b])$ maintains a balanced binary tree $\gamma(S[a, b])$ storing the edges of $\mathrm{CH}(F_{S[a,b]})$ in their cyclical ordering. We use this data structure as a black box, using the following functions that take at most $O(\log^2 n)$ time:

- $T(S[a, b])$.`get_hull()` returns the tree $\gamma(S[a, b])$.
- $T(S[a, b])$.`split(`v`)` returns, for $v \in [a, b]$, $T(S[a, v])$ and $T(S[v, b])$.
- $T(S[a, b])$.`split(`$T([S[b, c]])$`)` returns $T(S[a, c])$.
- $T(S[a, b])$.`update(`v`)` updates, for $v \in [a, b]$, the set S (deleting or inserting v).

3 Testing whether a set can be ε-covered by a single segment

We consider the following subproblem: given a parameter ε, a set S of n distinct integers, and the edges of $\mathrm{CH}(F_S)$ stored in a balanced binary tree, can we compute in $O(\log n)$ time whether there exists an ε-cover f of complexity 1? Formally, we seek a line ℓ of slope at least

1 such that all points in F_S lie within L_∞-distance ε of ℓ. Let L (resp. U) denote the set obtained by shifting each $p \in F_S$ downwards and rightwards (resp. upwards and leftwards) by ε, and adding the point $(\infty, -\infty)$ (resp. $(-\infty, \infty)$).

▶ **Lemma 6.** *Let ℓ be a line of slope at least 1. Then all points in F_S lie within L_∞-distance ε of ℓ if and only if ℓ lies below all points in U and above all points in L.*

Proof. Any line with positive slope lies above $(\infty, -\infty)$ and below $(-\infty, \infty)$. Consider a point $p \in F_S$ and the two corresponding points $l \in L$ and $u \in U$ and denote by C an axis-aligned square of radius ε centred at p. If ℓ lies below l then all points on ℓ left of l lie below C. If ℓ lies above u then all points on ℓ right of u lie above C. If ℓ lies above l and below u then because ℓ has positive slope, it must intersect C. The statement follows. ◀

▶ **Corollary 7.** *ℓ is an ε-cover of S iff it has a slope ≥ 1 and separates $CH(L)$ from $CH(U)$.*

Given CH(F_S), we can extract CH(L) and CH(U) in $O(\log n)$ time. Chazelle and Dobkin [6, Section 4.2] remark that, in the negative case, convex hull intersection testing can be modified to produce a separating line. In our setting, the hulls consist of segments with slope at least 1, and any such separator corresponds to an ε-cover. Thus, our problem reduces to the classical convex hull intersection problem and we are seemingly done.

However, the history of convex hull intersection testing is long and intricate. Both Chazelle and Dobkin [6] and Dobkin and Kirkpatrick [10] independently proposed the first $O(\log n)$-time algorithms. In 1987, Chazelle and Dobkin [5] presented a more detailed description of their method. Dobkin and Kirkpatrick revisited their own work in 1990 [11], proposing a unified $O(\log^2 n)$-time algorithm for polyhedron intersection, which O'Rourke later identified as incorrect [28]. He corrected the argument and provided a C-implementation. Further work by Dobkin and Souvaine [9] noted that earlier implementations lacked robustness. More recently, Barba and Langerman [2] observed that the community still lacked a complete, robust algorithm for polyhedral intersection. They proposed an alternative $O(\log n)$ algorithm based on polar transformations. Walther's master's thesis [35], supervised by Afshani and Brodal, implemented both this and earlier methods, but the source code is no longer available.

This 35-year history highlights the complexity and subtlety of convex hull intersection testing. Despite its history, no robust and modern $O(\log n)$-time implementation is available. Moreover, no published algorithm explicitly computes a separating line in the negative case.

Contribution. In the full version, we present a robust $O(\log n)$-time algorithm for convex hull intersection testing. Our algorithm is specialised to convex hulls composed of positively sloped segments and including the points $(\infty, -\infty)$ and $(-\infty, \infty)$. We formally prove its correctness and adapt it to compute a separating line in the negative case, thereby constructing an ε-cover of complexity 1 when it exists. The later adaption and its analysis are nontrivial, and arguably (partly) fill a gap in the existing literature on convex hull intersection testing.

▶ **Theorem 8.** *Let A and B be convex chains of edges with slope at least 1, stored in a balanced binary tree on their left-to-right order. There exists an $O(\log n)$ time to decide whether there exists a line that separates A and B. This algorithm requires only orientation-testing for ordered triangles and can output a separating line whenever it exists.*

4 Robustness

A geometric predicate is a function that takes geometric objects and outputs a Boolean. Our algorithms compute geometric predicates and use their output to branch along a decision tree. In F_S, consecutive points differ in x-coordinate by exactly 1 whilst their y-coordinate may wildly vary. Consequently, any segment that ε-covers a subsequence of F_S is quite steep. This quickly leads to rounding errors when computing geometric predicates, which in turn creates robustness errors. To illustrate our point, we discuss one of our main algorithms:

Algorithm 1 `intersection_test(edge` $\alpha \in CH(A)$, `edge` $\beta \in CH(B)$).

```
1:  if $\alpha = null$ OR $\beta = null$ then
2:      return No
3:  end if
4:  $s(\alpha, \beta) = line(\alpha) \cap line(\beta)$
5:  if If $s \in \alpha$ and $s \in \beta$ then
6:      return Yes
7:  end if
8:  if $\alpha$.slope < $\beta$.slope then
9:      if $\alpha$.first.$x > s(\alpha, \beta)$.x then
10:         return intersection_test($\alpha$.left, $\beta$)
11:     else if $\beta$.first.$x > s(\alpha, \beta)$.x then
12:         return intersection_test($\alpha$, $\beta$.left)
13:     else if $\alpha$.first.$x > \beta$.second.$x$ AND $\alpha$.first.$y > \beta$.second.$y$ then
14:         return intersection_test($\alpha$.left, $\beta$)
15:     else if $\alpha$.second.$x < \beta$.first.$x$ AND $\alpha$.second.$y < \beta$.first.$y$ then
16:         return intersection_test($\alpha$, $\beta$.left)
17:     else
18:         return yes
19:     end if
20: end if
21: if $\alpha$.slope > $\beta$.slope then
22:     if $\alpha$.second.$x < s(\alpha, \beta).x$ then
23:         return intersection_test($\alpha$.right, $\beta$)
24:     else if $\beta$.second.$x < s(\alpha, \beta).x$ then
25:         return intersection_test($\alpha$, $\beta$.right)
26:     else if $\alpha$.first.$x > \beta$.second.$x$ AND $\alpha$.first.$y > \beta$.second.$y$ then
27:         return intersection_test($\alpha$, $\beta$.right)
28:     else if $\alpha$.second.$x < \beta$.first.$x$ AND $\alpha$.second.$y < \beta$.first.$y$ then
29:         return intersection_test($\alpha$.right, $\beta$)
30:     else
31:         return yes
32:     end if
33: end if
```

`intersection_test` (Algorithm 1) which determines whether an upper quarter convex hull $CH(A)$ and a lower quarter convex hull $CH(B)$ intersect. We receive these hulls as two trees. Our algorithm computes a few geometric predicates given the edges α and β stored at their respective roots. Given (α, β), we either conclude that $CH(A)$ and $CH(B)$ intersect, or, that all edges succeeding (or preceding) α (or β) cannot intersect the other convex hull. Based on the Boolean output, our algorithm then branches into a subtree of α (or β). This way, we verify whether $CH(A)$ and $CH(B)$ intersect in logarithmic time. Rounding causes these predicates to output a wrong conclusion, and our algorithm may branch into a subtree containing edges of $CH(A)$ that are guaranteed to not intersect $CH(B)$. Our algorithm then wrongfully concludes that there exists a line ℓ separating $CH(A)$ and $CH(B)$. Subsequent algorithms then exhibit undefined behaviour when they attempt to compute this line.

Geometric predicates. Our algorithms use on three predicates for their decision making:

- `slope`. Given positive segments (α, β), output whether $\text{slope}(\alpha) < \text{slope}(\beta)$.
- `lies_right`. Given two positive segments α and β with different slopes, output whether the first vertex of β lies right of $line(\alpha) \cap line(\beta)$.
- `wedge`. Consider a pair of positive segments (α, γ) that share a vertex and define W as the cone formed by their supporting halflines containing $(\infty, -\infty)$. Given a positive segment β outside of W, output whether $line(\beta)$ intersects W.

The segments are given by points with integer coordinates. The slopes of these segments (and thereby any representation of their supporting line) are often not integer. A naive way to compute these predicates is to represent slopes using *doubles*. However, this is both computationally slow and prone to rounding errors (and thus, robustness errors).

If we insist on correct output, one can use an *algebraic type* instead. This type represents values using algebraic expressions. E.g., the slope of a positive segment (a, b) is the quotient: $\frac{b.y - a.y}{b.x - a.x}$ and so, in our case, it can be represented as a pair of integers. Algebraic types can subsequently be accurately compared to each other. Indeed, if we want to verify whether $\frac{s}{t} < \frac{q}{r}$ we may robustly verify whether $sr < qt$ using only integers. Exact (algebraic type) comparisons are frequently implemented, and present in the CGAL CORE library [12].

However, exact comparisons are expensive. Our implementation of `slope` requires two integer multiplications, which is still relatively efficient. Evaluating more complex expressions requires too much time. As a rule of thumb, we want to avoid compounding algebraic types to maintain efficiency. Naïvely, `lies_right` compounds two quotients and `wedge` compounds three. We give robust implementations of these functions by invoking three subfunctions. These compare slopes, or whether a point lies above or below a supporting halfplane:

$$\begin{aligned}
&\texttt{slope}((a,b),(c,d)) && := (b.y - a.y) \cdot (d.x - c.x) < (d.y - c.y) \cdot (b.x - a.x) \\
&\texttt{above\_line}((a,b),c) && := (b.x - a.x)(c.y - b.y) - (c.x - b.x)(b.y - a.y) \geq 0 \\
&\texttt{below\_line}((a,b),c) && := (b.x - a.x)(c.y - b.y) - (c.x - b.x)(b.y - a.y) \leq 0
\end{aligned}$$

We can create `lies_right` from our robust predicates (see Figure 3 (a)):

▶ **Lemma 9.** *Let $\alpha = (a, b)$ and $\beta = (c, d)$ be two positive segments of different slope. Then:*

$$\begin{aligned}
\texttt{lies\_right}(\alpha, \beta) = {} & \big(\texttt{slope}((a,b),(c,d)) == \texttt{above\_line}((a,b),c)\big) \\
& \vee \big(\texttt{slope}((c,d),(b,c)) == \texttt{below\_line}((a,b),c)\big)
\end{aligned}$$

Proof. Suppose that $\text{slope}(\alpha) < \text{slope}(\beta)$. Then c lies right of $line(\alpha) \cap line(\beta)$ if and only if c lies above the halfplane bounded from above by $line((a, b))$. That happens if and only if (a, b, c) are collinear or make a counter-clockwise turn. This in turn occurs if and only if the determinant if the matrix $\begin{vmatrix} (b.x - a.x) & (c.x - b.x) \\ (b.y - a.y) & (c.y - b.y) \end{vmatrix}$ is zero or more. If $slope(\alpha) > slope(\beta)$ the determinant must be negative instead. ◀

Similarly, we can create `wedge` from our robust predicates. We note for the reader that explain our equations in words in the proof of the lemma:

▶ **Lemma 10** (Figure 3 (b)). *If $\alpha = (a, b)$, $\gamma = (b, c)$ and $\beta = (d, e)$ be three segments of positive slope where $W = \overleftarrow{\alpha} \cup \overrightarrow{\gamma}$ bounds a convex area containing $(\infty, -\infty)$. Then*

$$\begin{aligned}
&\texttt{wedge}(\alpha, \gamma, \beta) := \\
&\big(\texttt{below\_line}((b,c),d) \wedge \big(\texttt{above\_line}((d,e),b) \vee \texttt{slope}((a,b),(d,e))\big)\vee \\
&\big(\texttt{below\_line}((a,b),e) \wedge \big(\texttt{above\_line}((d,e),b) \vee \texttt{slope}((d,e),(b,c))\big)\big)\vee \\
&\big(\neg\texttt{below\_line}((a,b),e) \wedge \neg\texttt{below\_line}((b,c),d) \wedge \big(\texttt{slope}((a,b),(d,e)) \vee \texttt{slope}((b,c),(d,e))\big)\big)
\end{aligned}$$

Proof. The predicate is a case distinction of three mutually exclusive cases.

If the first vertex of β lies below the supporting line of γ then $line(\beta)$ intersects W if and only if it intersects $\overleftarrow{\alpha}$. This happens if and only if one of two conditions hold: either b lies below the supporting line of b, or, slope$(\alpha) <$ slope(β).

If the second vertex of β lies below $line(\alpha)$ then the argument is symmetric.

If neither of those cases apply then both endpoints of β must lie in the open green area. In this case, whenever slope$(\alpha) <$ slope(β), the supporting line of β always intersects W. Whenever slope$(\beta) <$ slope(γ), the supporting line of β always intersects W. Whenever slope$(\alpha) \geq$ slope$(\beta) \geq$ slope(γ), the supporting line of β cannot intersect W. ◀

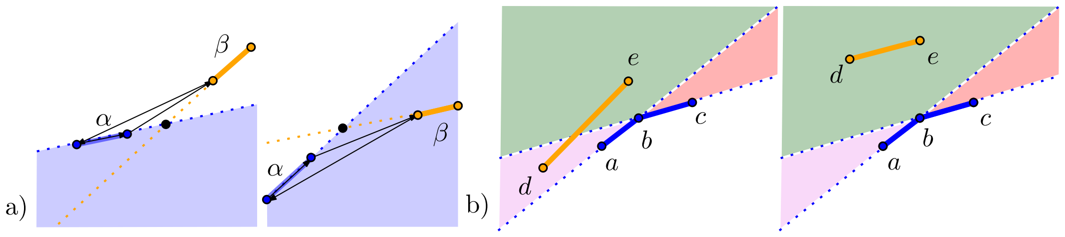

Figure 3 (a) We reduce testing whether the first vertex of β lies right of the intersection point to comparing slopes and the orientation of a triangle. (b) If d lies below the halfplane of $line(b, c)$ then $line((d, e))$ intersects the wedge if and only if b lies below $line((d, e))$.

5 Dynamically maintaining a learned index

We dynamically maintain a learned index h_ε of S by maintaining an ε-cover f of S. We guarantee that there exists no ε-cover f' of S with $|f| > 2|f'|$. By Observation 3, we obtain a learned index h_ε. By Observation 4, there exists no PGM index h_ε where $|f| > 2|h_\varepsilon|$.

To maintain f, we maintain a balanced binary tree $B(f)$ over $\Lambda(f)$. Additionally, for each $[a, b] \in \Lambda(f)$, we maintain a *rank-based convex hull* $T(S[a, b])$ of $S[a, b]$ as described in [17]. We note that we store all segments in f using *relative* x-coordinates. That is, we assume for all $[a, b] \in \Lambda(f)$ that the rank of the first element in $S[a, b]$ is zero. We may then use $B(f)$ to "offset" each line to compute the actual coordinates in rank-space.

▶ **Theorem 11.** *We can dynamically maintain an ε-cover f of S in $O(\log^2 n)$ worst-case time. We guarantee that there exists no ε-cover f' of S where $|f| > 2|f'|$.*

Proof. The proof is illustrated by Figure 4. For any $s, t \in \mathbb{Z}$ with $s \leq t$, we say that $S[s, t]$ is *blocked* if there exists no ε-cover of $S[s, t]$ of size 1. We maintain an ε-cover f where for all consecutive intervals $[a, b], [c, d] \in \Lambda(f)$, $S[a, d]$ is blocked. Thereby, $|f| \leq 2|f'|$ for any ε-cover f' of S (we give a proof of this fact in the full version).

We consider inserting a value s into S; deletions are handled analogously. We query $B(f)$ in $O(\log n)$ time for an interval $[a, b]$ that contains s. If no such interval exists, set $[a, b] = [s, s]$. We search $T(S[a, b])$ and test whether $s \in S$. If so, we reject the update.

Otherwise, we remove $[a, b]$ from $\Lambda(f)$ and insert the intervals $([a, s], [s, s], [s, b])$. We obtain $T(S[a, s])$, $T(S[s, s])$ and $T(S[s, b])$ through the split operation.

Let $([w, x], [y, z], [a, s], [s, s], [s, b], [c, d], [e, f])$ be consecutive intervals in $\Lambda(f)$ and denote $I = ([y, z], [a, s], [s, s], [s, b], [c, d])$ (see Figure 4 (c)). For each $(s, t) \in I$, we have access to $T(S[s, t])$. For any consecutive pair $([s, t], [q, r])$ in I, we may join the trees $T(S[s, t])$ and $T(S[q, r])$ in $O(\log^2 n)$ time to obtain $T([s, r])$. We invoke $T([s, r])$.`get_hull()` and apply

Theorem 8 to test in $O(\log^2 n)$ total time whether $S[s, r]$ is *blocked*. If it is not, we replace $[s, t]$ and $[q, r]$ by $[s, r]$. Otherwise, we keep $T(S[s, r])$ and a complexity-1 ε-cover of $S[s, r]$.

By recursively merging pairs in I, we obtain in $O(\log^2 n)$ time a sequence I' of intervals $([y, \beta], \dots, [\gamma, d])$ where consecutive intervals are blocked. Since $[y, z] \subseteq [y, \beta]$, $([w, x], [y, \beta])$ is blocked. Similarly, $([\gamma, d], [e, f])$ must be blocked. We remove the line segments corresponding to I from f and replace them with line segments derived from I' in constant time. As a result, we maintain our ε-cover f and our data structure in $O(\log^2 n)$ total time. ◀

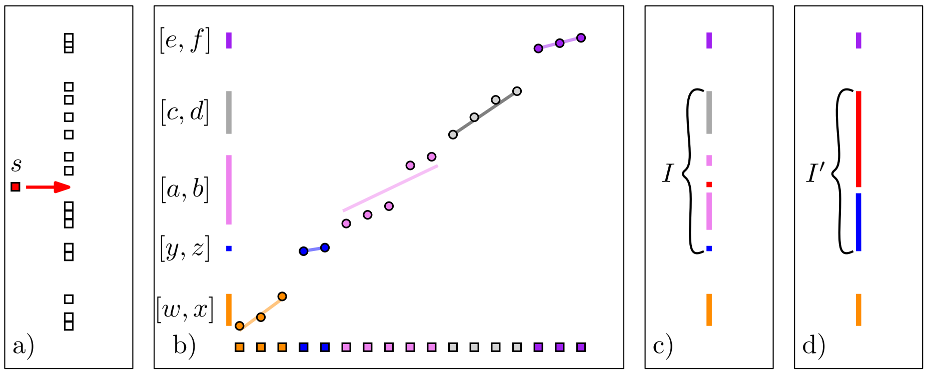

Figure 4 (a) Let S be a set of values and let us insert s. (b) We consider our ε-cover f and five consecutive intervals in $\Lambda(f)$. (c) We create seven intervals by splitting $[a, b]$ on s. (d) By recursively merging intervals in I, we obtain a set of intervals I' where consecutive intervals are blocked.

6 From an ε-cover to an indexing structure

A learned index h_ε does not immediately support indexing and range queries. We obtain an indexing structure by combining h_ε with a hash map H. Combining learned models with hash maps is not new [25, 34, 36] and this technique has even been applied to learned indexing [24]. The core idea is to store S in an unordered vector A and maintain a Hash map $H : \mathbb{Z} \mapsto [n]$. Given some $q \in \mathcal{U}$, the learned function then produces a value v such that $A[h(v)]$ is "close" to q. It is compelling to create H such that $A[H(h(q))]$ is (approximately) the predecessor of q. However, dynamically, this approach fails for the same reason that storing each $s \in S$ at $A[\texttt{rank}(s)]$ fails. Since the ranks of elements in S are constantly changing, we build a hash map using the parts of S that remain constant: the values.

Our data structure. In the full version, we define a data structure independent of the learned index h_ε (we illustrate our approach in Figure 5 (a) + (b)). A *page* p is an integer with a vector that stores all $s \in S$ where $\lfloor \frac{s}{\varepsilon} \rfloor = p$, in order. We store all non-empty pages P in an unordered vector A. We maintain a hash map $H : P \to [|A|]$, where $A[H(p)]$ contains the page p. We additionally maintain a doubly linked list over all pages in P, arranged in sorted order.

Our queries. We restrict our learned index h_ε to a *vertical ε-cover*. I.e., h_ε is a y-monotone collection of line segments such that for all points $p \in F_S$, a vertical line segment of height 2ε centred at p intersects a segment in h_ε. We compute h_ε oblivious of our paging structure.

Given $q \in \mathcal{U}$, we project q onto h_ε (Figure 5 (d)). We project to the x-axis, floor the value, and project back to h_ε. We prove that the resulting y-value corresponds to the page p containing `predecessor`(q). This way, we answer `predecessor` using $O(\varepsilon + \log |h_\varepsilon|)$ time.

As a result, we dynamically maintain a learned index h_ε and a data structure that updates in $O(\varepsilon + \log^2 n)$ and supports indexing queries in $O(\varepsilon + \log |h_\varepsilon|)$ expected time (Theorem 12).

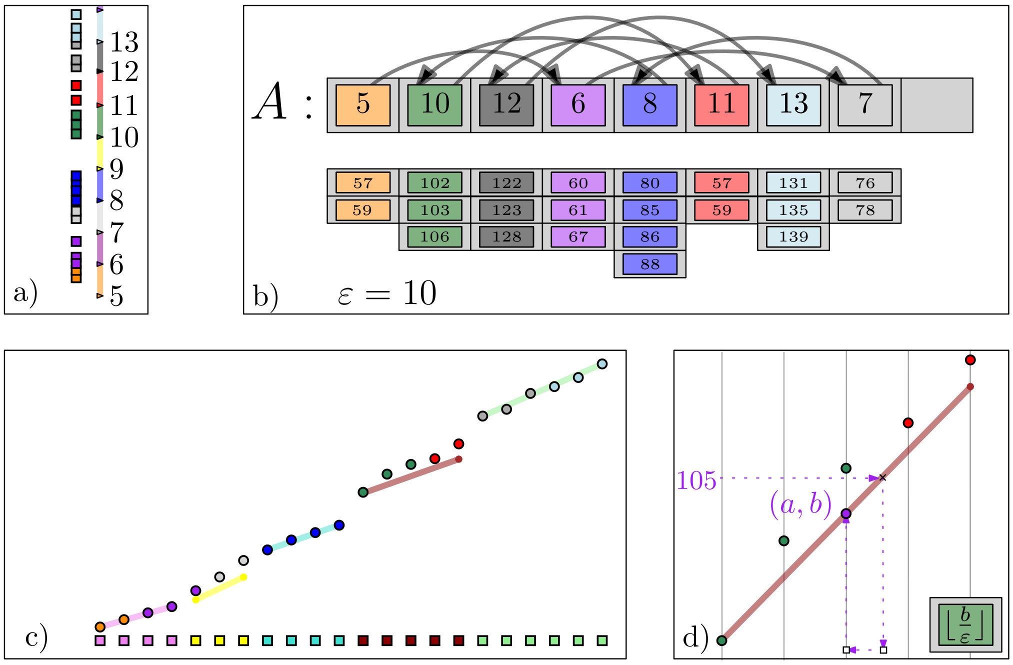

Figure 5 An illustration of our approach in the full version.

▶ **Theorem 12.** *For any ε, there exists a data structure to dynamically maintain a vertical ε-cover F of a dynamic set of distinct integers in $O(\varepsilon + \log^2 n)$ time. We guarantee that there exists no vertical ε-cover F' with $|F| > 2|F'|$. The data structure supports indexing queries in $O(\varepsilon + \log |F|)$ expected time and range queries in additional $O(k)$ time where k is the output size.*

7 Experiments

Our implementation is written in `C++` and made publicly available [18]. We compare to the `C++` implementation in [16], which uses a PGM index under the logarithmic method. The experiments were conducted on a machine with a 4.2GHz AMD Ryzen 7 7800X3D and 128GB memory. Our test bench is available [19], and can replicate experiments, generate synthetic data, and produce plots. As input we consider two synthetic data sets and two real world data sets. Three contain data of geometric nature, with one of random nature to align with precedent. Each set consists of unique 8 byte integers in randomly shuffled order. In the full version, we showcase additional experiments on other datasets.

- **LINES** is a synthetic data set of 5M integers that, in rank space, produces 5 lines of exponentially increasing slope. This set models the ideal scenario for a PGM index.
- **LONGITUDE** is a real world data set that contains the longitudes of roughly 246M points of interest from OpenStreetMap, over the region of Italy. This data is thereby inherently of geometric nature. This data set was used in both [16] and [21]. We follow [16] and convert the data to integers by removing the decimal point from the raw longitudes.

- **UNIF** originates from [16]. It is a synthetic data set, containing a uniform random sample of 50M integers from $(0, 10^{11})$. We adapt this data set to our dynamic setting.
- **DRIFTER** is a real world data set, containing roughly 1.7M steps of accumulated distance travelled by ocean drifters tracked through GPS [7].

Measurements. We compare the quality of the learned indices based the complexity of h_ε, in a dynamic setting. We use the same choice of $\varepsilon = 64$ as in [16] across our experiments. For performance of the indexing structures, we measure their time per operation in a dynamic scenarios with a range of query to update ratios. We note that logarithmic PGM is by default equipped with an optimisation that avoids building a PGM for data below a certain size. In this case, it instead only uses an underlying sorted array without additional search structure. In order to properly compare the performances, this optimisation has been disabled.

7.1 The learned index complexity

Figure 6 presents the complexity of the learned indices, measured by the number of line segments maintained during random-order insertions. The behaviour differs between datasets. For the geometric LINES and LONGITUDE datasets, the dynamic and logarithmic PGMs initially perform similarly. As the data grows, the performance of the logarithmic PGM degrades and its jagged progression reflects its logarithmic partitioning.

On the highly structured synthetic LINES data, the logarithmic method consistently retains more segments than necessary. It misses the optimal line count by a wide margin due to its fragmentation across $O(\log n)$ buckets. A similar pattern appears in the LONGITUDE dataset, where the logarithmic PGM maintains roughly 50 percent more segments than our solution. We note that precisely on these structured data sets, the complexity of the learned indices is $o(n)$. I.e., precisely here one also expects improvements in query time.

For unstructured data such as UNIF, both learned indices display similar asymptotic trends, with the logarithmic PGM using approximately 30 percent fewer segments. Here, the complexity appears to scale as $\Omega(n)$ for both methods, suggesting that learned indices offer few to no improvements in the absence of exploitable structure. Surprisingly, the DRIFTER data – despite its geometric origin – shows similar results to that of UNIF. Again, complexity scales linearly, and the logarithmic method outperforms the dynamic one by around 30 percent. This implies that whatever latent geometric structure exists is insufficiently captured by the learned index under either strategy.

An interesting observation is that, on unstructured data, the imposed bucketing of the logarithmic PGM index can introduce a form of regularity that the model benefits from, essentially imposing artificial structure where none exists.

7.2 Running time comparisons

Recall that our indexing structure is composed of a learned index over a vertical ε-cover and a paging structure. We first examine the performance of both the learned index in its own, and then the performance of the full indexing structure.

For the dynamic scenarios, we first follow the precedence set by prior papers [16, 21]. These first construct, insertion-only, the indexing structure. They then perform a batch of 10M operations These batches consist of insertions, deletions, and range queries over ranges such that the output contains approximately $\frac{\sqrt{n}}{10}$ elements. We deviate from the precedent by also deleting from the index before performing a batch of operations, to simulate a scenario in which the structure has existed and transformed prior to processing.

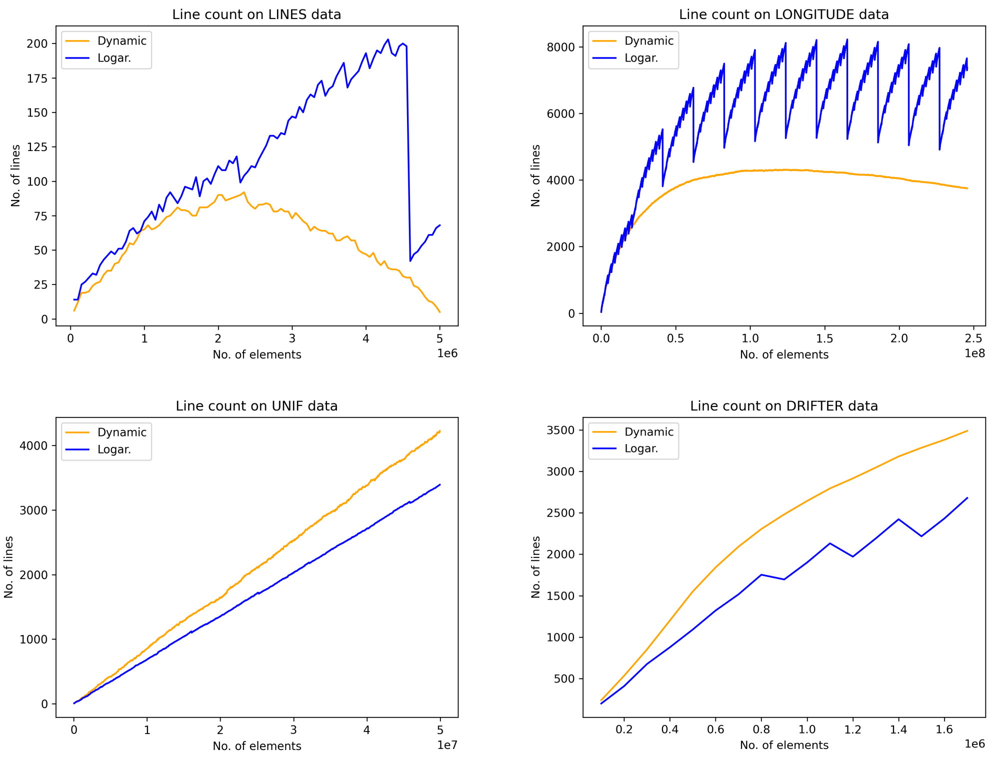

Figure 6 The complexity of the learned indices throughout insertion-only construction. The top graphs represent geometrically structured data. The left graphs represent synthetic data.

7.2.1 Maintaining a learned index

Figure 7 shows the cost of maintaining each learned index under dynamic operations. Our update procedure, which ensures worst-case $O(\log^2 n)$ bounds, performs significantly worse than the amortised $O(\log n)$ updates offered by the logarithmic method – especially on larger datasets such as LONGITUDE and UNIF. On smaller datasets like DRIFTER, the gap narrows, but the logarithmic method generally remains preferable in these update-only scenarios.

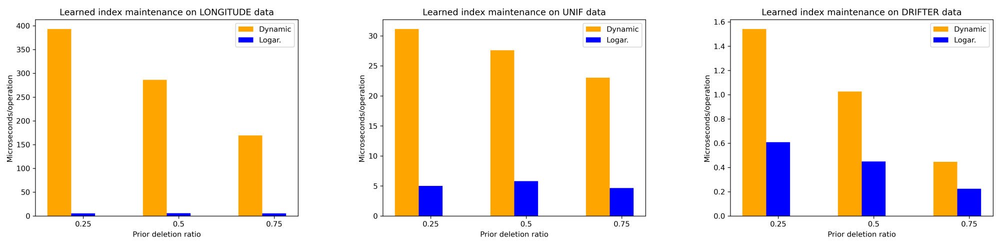

Figure 7 Update times for maintaining the learned index dynamically.

7.2.2 Indexing data structures

We next assess the complete indexing structures under mixed workloads. Batches contain a tunable ratio of queries and updates, where updates are evenly split between insertions and deletions, and operations are randomly ordered. More experiments can be found in the full version. For the LINES dataset, updates are constrained to a single line segment to preserve its idealised structure. For smaller datasets, batch size is limited to 25 percent.

Adversarial workload (ADV). In the above scenario by [16, 21], an update batch of $10M$ operations affects less than ten percent of the data. Therefore, we do not encounter the worst case scenario where range queries take $O(N + \varepsilon + \sum_i^{\lceil \log n \rceil} \log |f_i|)$ time. So, the worst case difference in performance does not come to light. Therefore, we additionally construct an *adversarial scenario* consisting of 10M range queries after deleting all but 1.000 values.

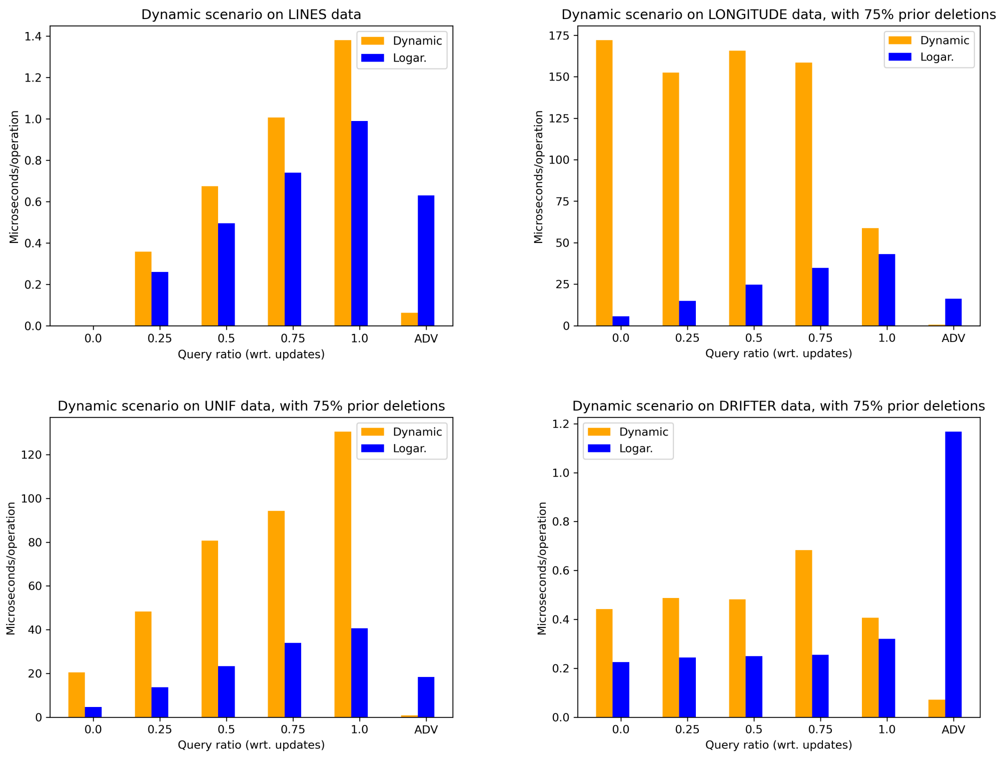

Figure 8 Time per operation in dynamic scenario with varying query ratios. For the LINES data set updates are restricted to points on a single line. The ADV ratio denotes the adversarial case where all but a constant number of elements are deleted before the structure is queried.

Operational performance. Figure 8 summarises the total processing time under varying query-to-update ratios. Structured datasets, particularly LONGITUDE, reveal a trade-off between the fact that we have a lower-complexity learned index and our update time. On one hand, our updates are costly due to our line-merging tests which suffer from the poor cache behaviour from pointer-based trees, and complex rebalancing that is present in the state-of-the-art practical dynamic convex hull data structures. This behaviour is also reflected on our previous analysis of maintaining the learned index itself. On the other hand we see on geometrically structured data that, as the query ratio increases, performance improves. This trend is absent in the Lines data, where updates are constrained and the structure is small, mitigating cache penalties. Here, running time is largely driven by query processing, which increases as expected with query ratio.

In contrast, the random data from the Unif dataset shows improved performance with higher query ratios. In our data structure, having smaller segments reduce restructuring costs. Our dynamic implementation suffers greatly from cache inefficiency due to random access during range reporting. The logarithmic PGM, with its lower memory overhead and sequential layout, delivers significantly better performance – particularly as sizes increase.

For the Drifter data, one would expect a trend similar to that of the Unif data, based on the complexity of the ε-cover shown in Figure 6. However, there is little difference in performance for either PGM as the query ratio is varied. This is likely due to the small size of the data set, with both structures performing updates in fractions of microseconds on average. The logarithmic PGM does come out slightly on top, likely due to the machine friendly memory access pattern.

Adversarial impact. Across all datasets, the logarithmic PGM's tombstoning strategy becomes a bottleneck in the adversarial scenario. Its range queries must scan and subtract deleted values before outputting results. Our structure, by contrast, is output-sensitive and avoids such overhead. This illustrates that our data structure, whilst being generally less efficient that the logarithmic PGM, offers worst-case guarantees.

8 Conclusion

We studied dynamic learned indices through a geometric perspective. Following the work of Ferragina and Vinciguerra [16], we maintained a learned index h_ε of a dynamic set S as a piecewise-linear approximation – an ε-cover – of the rank-space point set F_S. We used techniques from computational geometry to answer the following question:

"Can a learned index be dynamically maintained with worst-case guarantees?"

We proposed a new approach to maintain a learned index based on dynamic convex hull data structures. We presented an $O(\log^2 n)$ time algorithm to dynamically maintain a learned index h_ε. To obtain this algorithm, we showed an algorithm to compute a separating line between two non-intersecting convex hulls – an operation previously missing from the literature. The existing logarithmic PGM index has an amortised $O(\log n)$ time update algorithm which is more efficient in practice. Indeed, although close in theory, the memory access pattern associated with dynamically maintaining convex hulls incurs heavy penalties for large datasets. At the same time, the resulting learned index of the logarithmic PGM, h'_ε, can be considerably more complex on geometrically structured data.

From learned indices to indexing. Finally, we considered the following question:

"Can a dynamic learned index be converted into a dynamic indexing data structure?"

To this end, we designed a hybrid technique combining hashing-based fast-access data structures with a doubly linked list to support indexing queries. Our method offers output-sensitive worst-case guarantees, even in the presence of deletions. As it is known that traditional indexing structures currently outperform learned indices in the general dynamic setting [33], we focused our comparisons on improving the theoretical and practical performance within the class of learned approaches.

In practice, the contiguous memory access of the logarithmic PGM index offsets its overhead from querying multiple structures, making it faster in all scenarios, except for an adversarial one. This means that the lower complexity of our learned index does not immediately translate to improved efficiency in the indexing data structure. This raises an interesting open question of whether memory-access efficient fully dynamic approaches to convert a learned index into a dynamic indexing structure can exist.

While we acknowledge that our update-times are slow in comparison with state-of-the art, our approach does illustrate that it brings worst-case guarantees: as it has an advantage when the query-to-update ratio is large and the index has undergone sufficiently many deletions. In adversarial workloads with frequent deletions followed by range queries, we have seen our structure outperform the logarithmic approach – highlighting the value of worst-case guarantees even in specialised settings.

Closing thoughts. We showed what we believe is an interesting connection between the geometric learned index by Ferragina and Vinciguerra, and dynamic convex hulls from computational geometry. We subsequently provided an implementation of a dynamic learned index that relies on the state-of-the-art dynamic convex hull maintenance algorithms. Our empirical analysis shows that the complex tree rebalancing that is used to dynamically maintain a convex hull currently brings considerable operational overhead compared to low-memory techniques under the logarithmic method. Our experiments, though not uniformly favourable, offer interesting insights into the current barriers and adversarial tradeoffs between worst-case dynamic algorithms and memory-efficient amortised rebuilding schemes.

References

1 Manos Athanassoulis and Anastasia Ailamaki. Bf-tree: approximate tree indexing. In *International Conference on Very Large Databases (VLDB)*, 2014.

2 Luis Barba and Stefan Langerman. Optimal detection of intersections between convex polyhedra. *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2015.

3 Michael A Bender, Erik D Demaine, and Martin Farach-Colton. Cache-oblivious b-trees. In *Symposium on Foundations of Computer Science (FOCS)*. IEEE, 2000.

4 Chee-Yong Chan and Yannis E Ioannidis. Bitmap index design and evaluation. In *ACM International Conference on Management of Data (SIGMOD)*, 1998.

5 B. Chazelle and D. P. Dobkin. Intersection of convex objects in two and three dimensions. *Journal of the ACM*, 1987. `doi:10.1145/7531.24036`.

6 Bernard Chazelle and David P Dobkin. Detection is easier than computation. In *ACM Symposium on Theory Of Computing (STOC)*, 1980.

7 Jacobus Conradi and Anne Driemel. Finding complex patterns in trajectory data via geometric set cover. *CoRR*, abs/2308.14865, 2023. `doi:10.48550/arXiv.2308.14865`.

8 Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, et al. Alex: an updatable adaptive learned index. In *ACM International Conference on Management of Data (SIGMOD)*, 2020.

9 David Dobkin and Diane Souvaine. Detecting the intersection of convex objects in the plane. *Computer Aided Geometric Design*, 1991. `doi:10.1016/0167-8396(91)90001-R`.

10 David P. Dobkin and David G. Kirkpatrick. Fast detection of polyhedral intersection. *Theoretical Computer Science (TSC)*, 1983. `doi:10.1016/0304-3975(82)90120-7`.

11 David P. Dobkin and David G. Kirkpatrick. Determining the separation of preprocessed polyhedra - a unified approach. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 1990.

12 Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of cgal a computational geometry algorithms library. *Software: Practice and Experience*, 30(11):1167–1202, 2000. `doi:10.1002/1097-024X(200009)30:11\%3C1167::AID-SPE337\%3E3.0.CO;2-B`.

13 Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. Why are learned indexes so effective? In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 3123–3132. PMLR, 2020. URL: `http://proceedings.mlr.press/v119/ferragina20a.html`.

14 Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. On the performance of learned data structures. *Theor. Comput. Sci.*, 871:107–120, 2021. `doi:10.1016/J.TCS.2021.04.015`.

15 Paolo Ferragina and Giorgio Vinciguerra. Learned data structures. In *Recent Trends in Learning From Data: Tutorials from the INNS Big Data and Deep Learning (INNSBDDL)*. Springer, 2020.

16 Paolo Ferragina and Giorgio Vinciguerra. The pgm-index: a fully-dynamic compressed learned index with provable worst-case bounds. *International Conference on Very Large Databases (VLDB)*, 2020.

17 Emil Gæde, Inge Li Gørtz, Ivor Van Der Hoog, Christoffer Krogh, and Eva Rotenberg. Simple and robust dynamic two-dimensional convex hull. *ACM Symposium on Algorithm Engineering and Experiments (ALENEX)*, 2024.

18 Emil Toftegaard Gæde, Ivor van der Hoog, Eva Rotenberg, and Tord Stordalen. A Dynamic Piecewise-Linear Geometric Index with Worst-Case Guarantees. Software, Carlsberg Fonden CF21-0302, Villum Fonden VIL37507, Marie Skłodowska-Curie 899987, swhId: `swh:1:dir:b8763eb0504d33beb81ee89d230a30dca8ab0b66` (visited on 2025-09-03). URL: `https://github.com/Sgelet/DynamicLearnedIndex`, `doi:10.4230/artifacts.24667`.

19 Emil Toftegaard Gæde, Ivor van der Hoog, Eva Rotenberg, and Tord Stordalen. Testbed for our learned index repository. Software, Carlsberg Fonden CF21-0302, Villum Fonden VIL37507, Marie Skłodowska-Curie 899987, swhId: `swh:1:dir:07ea25cfc176438933c1a5507bfdad3ba9461ab6` (visited on 2025-09-03). URL: `https://github.com/Sgelet/LearnedIndexBench`, `doi:10.4230/artifacts.24668`.

20 Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *ACM International Conference on Management of Data (SIGMOD)*, pages 1189–1206, 2019. `doi:10.1145/3299869.3319860`.

21 Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Sosd: A benchmark for learned indexes. *Conference on Neural Information Processing Systems (NEURIPS)*, 2019.

22 Andreas Kipf, Ryan Marcus, Alexander van Renen, Mihail Stoian, Alfons Kemper, Tim Kraska, and Thomas Neumann. Radixspline: a single-pass learned index. In *International workshop on exploiting artificial intelligence techniques for data management*, 2020.

23 Nick Koudas. Space efficient bitmap indexing. In *ACM international conference on Information and knowledge management (SIGMOD)*, 2000.

24 Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. The case for learned index structures. In *ACM International Conference on Management of Data (SIGMOD)*, 2018.

25 Yuming Lin, Zhengguo Huang, and You Li. Learning hash index based on a shallow autoencoder. *Applied Intelligence*, 2023.

26 Ryan Marcus, Andreas Kipf, Alexander van Renen, Mihail Stoian, Sanchit Misra, Alfons Kemper, Thomas Neumann, and Tim Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, 2020. `doi:10.14778/3421424.3421425`.

27 Joseph O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 1981.

28 Joseph O'Rourke. *Computational geometry in C (second edition)*. Cambridge University Press, USA, 1998.

29 Mark H Overmars. *The design of dynamic data structures*, volume 156. Springer Science & Business Media, 1983. doi:10.1007/BFB0014927.

30 Mark H Overmars and Jan Van Leeuwen. Maintenance of configurations in the plane. *Journal of computer and System Sciences*, 1981.

31 Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 2004.

32 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *ACM Symposium on Theory of Computing (STOC)*, 2006.

33 Zhaoyan Sun, Xuanhe Zhou, and Guoliang Li. Learned index: A comprehensive experimental evaluation. *Proc. VLDB Endow.*, 16(8):1992–2004, 2023. doi:10.14778/3594512.3594528.

34 Antonio Torralba, Rob Fergus, and William T Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence*, 2008.

35 Lukas Walther, Gerth Brodal, and Peyman Afshani. Intersection of convex objects in the plane. Master's thesis, Aarhus University, 2015. Available at https://cs.au.dk/~gerth/advising/thesis/lukas-walther.pdf.

36 Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. Learning to hash for indexing big data—a survey. *Proceedings of the IEEE*, 2015.

37 Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G Andersen. Building a bw-tree takes more than just buzz words. In *ACM International Conference on Management of Data (SIGMOD)*, 2018.

38 Chaichon Wongkham, Baotong Lu, Chris Liu, Zhicong Zhong, Eric Lo, and Tianzheng Wang. Are updatable learned indexes ready? *International Conference on Very Large Databases (VLDB)*, 2022.