Deconstructed "Learned" Indexes and Their Smoothed Analysis

Stefan Hermann
Karlsruhe Institute of Technology, Germany

Mattia Odorisio
University of Pisa, Italy

Peter Sanders
Karlsruhe Institute of Technology, Germany

Stefan Walzer
Karlsruhe Institute of Technology, Germany

Abstract

Data structures that maintain a sorted sequence are crucial for many applications. There is a zoo of variants with recent particular interest in "learned" indexes that accelerate operations by learning the distribution of the data. This paper helps to bring some order to this complex situation. We identify important building blocks and model the input using smoothed analysis where an adversary can control the dynamically changing input except for a small amount of noise. Within a resulting design space of data structures, we prove that already a simple 2-level data structure with minimal learning can achieve constant operation times in many situations: PARROT partitions the input into equal size parts, within which keys are approximately uniformly distributed. In many of our experiments, PARROT performs very well compared to state-of-the-art learned indexes, being 2× faster than the well known ALEX and LIPP indexes on large datasets, and 10× faster than a well engineered standard B-Tree.

2012 ACM Subject Classification Information systems → Data structures; Information systems → Database management system engines; Computing methodologies → Machine learning

Keywords and phrases Learned data structure, sorted sequence, index data structure, smoothed analysis

Digital Object Identifier 10.4230/LIPIcs.ESA.2026.15

Supplementary Material *Software*: `https://github.com/mattiaodorisio/PARROT-testbed`
archived at `swh:1:dir:371700bcc9bdc0542bf52ef91331d6512ddb2337`
Software: `https://github.com/mattiaodorisio/PARROT`
archived at `swh:1:dir:f0f99eef3ad82c32085434f2010580d858ca805a`

1 Introduction

Sorted sequences $\langle x_1, \ldots, x_n \rangle$, are a fundamental building block underlying many commonly used containers such as dictionaries, ordered (multi)sets, or maps. They can either be stored directly by using a data structure such as a search tree, or augmented with indexes to offer efficient query functionality over the data. Many concrete solutions have been developed. See [45, 2] for survey papers and Section 3 for a summary. When the data structure is based on comparing keys, most operations (like insert, delete, or find successor) take time $\Theta(\log n)$. Faster solutions are possible when more information from the keys is used (e.g., using multiple key bits for addressing arrays). For example, van Emde Boas search trees

(vEBs) support the basic operations in time $O(\log \log n)$ [56, 43].[1] It can also be helpful if the distribution of the data is known. In particular for uniform random data, expected constant time is possible. For example, to store n random numbers in the range $1..U$, we can maintain an array of n *buckets*, each responsible for storing keys in a range of size U/n. This results in buckets of constant expected occupancy.

Recently, there has been considerable interest in generalizing the above observation. The idea is to "learn" the distribution of more general inputs to support fast operations, ideally in constant time (e.g., [34, 15, 14]). Our starting point for this paper was, on the one hand, admiration for good practical performance and, on the other hand, frustration with weak theoretical results. Most previous results either do not give better guarantees than already available for van Emde Boas trees (which need fewer assumptions on the input) or make very strong assumptions that are not warranted in many cases. In particular, the important case of a dynamically changing distribution is often neglected. Moreover, previous learned indexes sometimes use complex multi-level data structures and it was not clear to us whether this is really necessary.

In this paper, we therefore want to start a process of "deconstructing" (learned) indexes, i.e., taking them apart to understand their essential features and "reconstructing" new, streamlined, better understood data structures by putting the pieces together in new ways. Overall, this opens up a considerable design space.

Rather, in section Section 4 we consider a simple 2-level learned index PARROT (**PAR**titioned **RO**bin Hood **T**able) as an example. At the root level, PARROT applies a simple linear transformation consisting of an additive offset and a bit-shift. The resulting nonnegative integer is split into two segments. The most significant bits index a top-level hash table of nonempty tables on the bottom level. Bottom-level tables are organized as adaptively growing and shrinking hash tables that are directly indexed by the least significant bits of the key. Successor and predecessor queries are optionally aided by two levels of bitvectors (or by a vEB in the theoretical analysis). In Appendix B we outline further results from the design space on using B-Trees, static indices, and priority queues.

For analyzing the data structures, we adapt the concept of *smoothed analysis* which was previously used to understand why simple algorithms for complex problems like linear programming [51] or knapsack [6] are efficient in practice. Concretely, we assume an adversary who presents a sequence of operations whose keys are perturbed with Gaussian[2] noise, i.e., when the adversary presents key x, the data structure actually uses a key drawn from the normal distribution $\mathcal{N}(x, \sigma^2)$ for some standard deviation σ. Parameter σ controls the magnitude of perturbation so that for very small σ, we are approaching a worst case regime. For sufficiently large σ we will get a smooth overall distribution that is quite easy to handle. In some applications, we can justify the perturbation by the fact that the keys stem from noisy sensors. In those cases, carefully crafted worst case inputs would be highly unlikely. A crucial advantage of our model is that it is dynamic by design. In contrast, much previous theoretical work is based on global distributions that are in some sense smooth. Then it becomes complicated to define how these distributions change. Section 5 describes our analytical results. Roughly, PARROT achieves constant expected time for operations on the bottom level when σ is larger than the width of the bottom-level tables. The same applies for in-distribution queries, long bursts of the same operation type, and for windows moving over a data set that only remove keys that have been inserted a long time ago.

---

[1] More precisely, operations take time $O(\log w)$ on keys of length w. However, the standard assumption in the RAM model of computation [49] is that inputs of size n are processed using word size $w = \Theta(\log n)$ so that we can characterize complexity also by input size.

[2] Other types of noise are conceivable. We adopt Gaussian noise as this is the most traditional assumption.

In Section 6 we compare PARROT with state-of-the-art learned indexes and show that they compete well in many cases. In particular, the dynamic version of PARROT outperforms static learned indexes on all synthetic distributions, and is 2× faster than well known ALEX and LIPP indexes on large datasets. It is also 10× faster than a well-engineered standard B-Tree, and has competitive performance with highly engineered predecessor data structures. We summarize and discuss lines of further research in Section 7.

Summary of contributions

- Analyzing essential components of index data structures such as B-Tree-nodes, arrays indexed by keys, bit-vectors, or hash tables (deconstruction) and explaining new ways to put them back together in useful ways (reconstruction).
- Using smoothed analysis as a natural model of dynamic real-world data.
- Design and analysis of a simple 2-level learned index PARROT that achieves constant-time operations in many situations.
- Experiments showing that PARROT works well in practice.

2 "Deconstruction" and Preliminaries

For the theoretical discussion, we consider a sorted sequence $S = \langle x_1, \ldots, x_n \rangle$ of keys. Our implementation also allows key-value pairs. We mainly consider the operations insert(x), delete(x), and successor$(x) := \min\{y \in S : x \leq y\}$ (assuming a sentinel key ∞). Other operations are similar (like predecessor or find) or can be constructed from these basic operations (e.g., range queries as a series of successor operations). Below, we review existing approaches for maintaining sorted sequences, viewing them not as monolithic data structures but as a source of building blocks that we can later reconstruct for different purposes.

B-Trees

Comparison-based sorted sequences are often implemented as a search tree. The search descends this tree using pivot keys stored at the tree nodes. There are many balancing strategies. B-Trees achieve uniform depth of all leaves by allowing node degrees in the range $a..b$ [5]. B-Trees are perhaps the most successful search trees because they allow some cache locality when degree b nodes fit into a cache line. We argue that B-Tree nodes may also be useful in reconstructing other data structures to handle parts of the input with very heterogeneous distribution. For example, suppose the key range includes $-\infty$ and ∞. Then a B-Tree node of degree ≥ 3 could take care of a case distinction necessary to zoom in on the more homogeneous other keys.

Van Emde Boas (vEB) trees and bit vectors

vEB-trees [56, 43] are a recursive data structure that splits a set of w-bit keys into subsets. Each subset contains keys that share the same most significant $w/2$ bits. A hash table and another recursive "top"-vEB-tree maintain the nonempty subsets. An engineered variant [18] keeps the top data structure in multiple layers of bit vectors. Layer 0 indicates the nonempty subsets and a bit in layer $i + 1$ is the logical-or of several bits in layer i.

Arrays for near-uniformly distributed data and (non)linear transformations

A uniformly distributed random key x can be mapped to an array cell (bucket) $a[\lfloor \alpha x + \beta \rfloor]$ using a linear transformation. It is well known that this yields constant expected query time if we scale the array and the transformation so that the expected bucket size is constant. Concretely, one can use an indirection to store a bucket elsewhere. We will save this indirection by viewing the linear transformation as a hash function into a linear probing hash table. Specifically, *Robin-Hood hashing* keeps the keys sorted by their hash function value [3, 12, 63]. By using a nonlinear transformation (e.g. [33]) one can adapt even better to the distribution.

3 Related Work

It has long been known that *interpolation search* [46] can search uniformly distributed data in expected time $O(\log \log n)$ [63]. An insightful hybrid with binary search [58] generalizes this to more general distributions that "look" locally uniform. Perhaps the most advanced result in this line of research [32] allows constant query time for dynamic data sets for certain "benign" static distributions. It is based on a 2-level data structure with an array at the top level and a sophisticated word-parallel data structure at the bottom level that can handle sets of polylogarithmic size in constant time. However, it seems that interpolation search was rarely used in practice, perhaps because the involved constant factors diminish their impact. vEBs achieve $O(\log w) = O(\log \log n)$ operations for worst case inputs [56, 43] and do perform well for 32 bit keys [18].

Learned indexes can be viewed as taking basic ideas from interpolation search and combining them with a learning perspective [34, 42, 53, 19, 60, 54, 61]. There are several quite successful practical implementations. However, efforts to analyze them theoretically [25, 24, 39, 64, 65] usually do not break the $\log \log n$ bound already achieved for worst case inputs by vEBs. There are also results showing constant query time. However, as far as we know, they only work for distributions with fairly large standard deviation. For example, Croquevielle et al. [15] analyze a static index (ESPC) that is similar to a variant of PARROT with only the top level and without bit vectors. This suffices to achieve constant expected query time when the probability distribution is very smooth (in our smoothed analysis model, this would apply for constant σ).

Analysis of dynamic scenarios is also rare and often involves complicated formalisms and considerable penalties for quickly changing distributions. For example, Zeighami and Shahabi [65] investigate dynamic learned indexes where query time varies between $O(\log \log n)$ and $O(\log n)$ depending on the allowed variation.

Out of the many available learned indexes we now discuss a few successful ones that also inspired PARROT: ALEX [19] is one of the most successful dynamic learned indices. At first glance, it is similar to a multilevel generalization of PARROT that learns a linear transformation at each node of the tree. However, ALEX does not use Robin Hood tables but *gapped arrays* [30], i.e., sorted arrays with free slots that allow binary/exponential search but without a clear association between estimated and actual position of a key. This results in expected search time logarithmic in the array size even for uniform distributions. Moreover, ALEX only has logarithmic insertion time guarantees when the insertion order is random [8].

PGM [25, 39] is perhaps the theoretically best understood learned index. It performs linear regressions with bounded error in a bottom up fashion. It seems that this approach makes it difficult to get better than $O(\log \log n)$ queries. This also makes a natively dynamic version difficult. Rather, dynamic PGM uses a hierarchy of insertion buffers (PGMs themselves) that incur additional query overhead. Refer to Appendix A for additional related work.

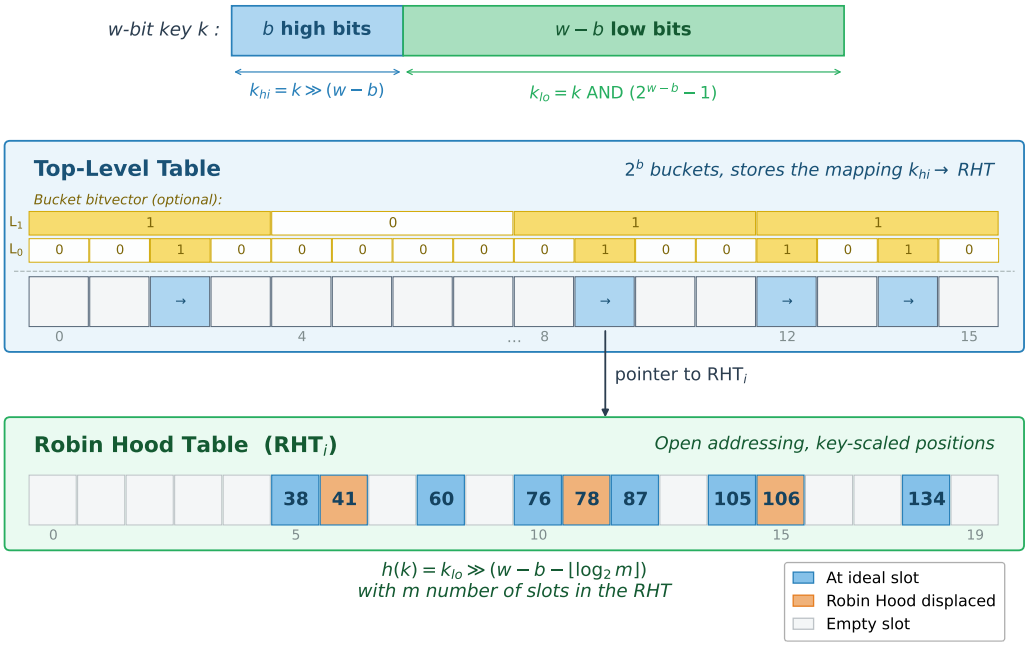


Figure 1 Overview of PARROT's two-level architecture. The b high-order bits of a key address the top level (blue box), where a two-level bitvector (yellow) enables fast routing to the nearest non-empty bucket. The remaining low-order bits then index into the corresponding Robin Hood table (RHT) at the bottom level (green box).

4 Design of PARROT

PARROT is a two-level predecessor data structure built around theoretical results presented in Section 5. Its design relies on partitioning the key space into narrow ranges: within each range, a linear function predicts where a key is stored. Concretely, the b most significant key bits k_{hi} index the top level. The remaining $w - b$ bits k_{lo} are the index for the bottom level. We use $h(k) = k_{\text{lo}} \gg w - b - \lfloor \log_2 m \rfloor$ to map a key to its slot, where m is a power of two describing the number of slots of the table. We dynamically change m to stay close to a targeted load factor. While any hash table could be used, we assume a linear probing table that keeps the keys sorted. This is known as a *Robin Hood Table* (RHT). What is actually learned here is $\lfloor \log_2 m \rfloor$, i.e., very little. Both upper and lower levels of PARROT can be equipped with vEB to aid predecessor/successor queries. Our analysis suggests that this is only needed in the upper level due to smoothing. Our implementation streamlines this to two levels of bit vectors. Figure 1 gives an example.

Queries. A successor query for a key q proceeds in two steps: *bucket routing* and *in-bucket resolution*. In the *bucket routing* phase, the top level is queried with the b high bits of q to retrieve the corresponding RHT. If no bucket exists for those high bits (i.e., no stored key shares that prefix), the vEB or the bitvectors are consulted to efficiently locate the nearest non-empty bucket to the right. Then, in the *in-bucket resolution* phase, starting from slot $h(q)$, the RHT is probed until the first key $\geq q$ is found. Successor queries are similar except that the RHT may be probed backwards.

Updates. Analogously to queries, an update for a key k begins with a *bucket routing* phase using the b high bits of k to locate the target bucket, possibly creating a fresh RHT if that bucket does not yet have one. The new key is placed near $h(k)$ with linear probing. Deletion similarly consists of locating the key as for queries, and removing it from the RHT, leaving a vacated slot. The key(s) stored immediately to the right of the deleted one are possibly left-shifted to fill the vacated slot depending on their $h(\cdot)$ so as to restore the probe sequence invariant.

Implementation details. The implementation incorporates the following design details. Both the top level and the bottom level store their keys using the narrowest available native unsigned integer type that can hold the required number of bits, keeping the memory footprint small while still allowing keys to be handled as ordinary integers. The top level is implemented as a hash table rather than a plain array to reduce space consumption (and cache efficiency) when choosing a large value for b which may be needed to make the bottom levels sufficiently uniform. As previously noted, the theoretical van Emde Boas tree at the top level is replaced by two levels of bit vectors to improve simplicity and performance. RHTs are searched in an SIMD-parallel way. There are variants of PARROT with and without values associated with the keys. In the former case, values are stored in a separate array in order to accelerate SIMD-search. A static variant of PARROT prefills empty slots at even (odd) positions with their predecessor (successor) keys. This accelerates predecessor (successor) queries by eliminating the need to probe neighboring empty slots. The experimental evaluation is presented in Section 6.

5 Analysis of PARROT

In this section we show that PARROT achieves expected constant time operations under certain assumptions. For notational convenience we assume that the keys are real values in $[0, 1]$. Let δ be the width of the buckets, i.e. there are $\frac{1}{\delta}$ buckets and a key x is mapped to bucket $\lfloor \frac{x}{\delta} \rfloor$.

Summary of the analysis. We assume *smoothed* inputs, i.e. the adversary specifies a real valued key $x \in [0, 1]$ but our data structure actually inserts a key sampled from $\mathcal{N}(x, \sigma^2)$. The time for an operation is then a random variable and we are interested in its expectation. If σ is at least the width of a bucket δ, then the smoothed keys are roughly uniformly distributed within the buckets. We then use standard results on linear probing to show constant expected operations on the RHTs. The smoothing therefore makes the keys locally, i.e., on the bottom level, harmless (Lemma 7). On the top level we use a vEB tree with an operation cost of $O\big(\log\log \frac{1}{\delta}\big)$. Without further assumptions, the time for operations on PARROT is the time for operations on the vEB (Proposition 8). We have to perform a vEB operation if the result for a query of x is not found within the bucket of x. In case of insertions (or deletion) of x, we have to perform a vEB insertion (deletion) if x is the first (last) key within its bucket. We consider three models in which vEB operations do not dominate, resulting in expected constant time operations on PARROT:

- *Burst* (Theorem 9). A burst describes a sufficiently large number of identical operations. For queries, we can then afford to precompute the predecessor and successor of each bucket. For insertions and deletions we can rebuild the vEB after the burst.
- *In-distribution* (Theorem 10). In the in-distribution model we assume that operations are performed on inputs that look similar to the data that is already in the index.

- *Sweep-line* (Theorem 11). In this model, we assume that only keys may be deleted that have been in the data structure for sufficiently long, i.e. a lot of other insertions have occurred meanwhile. This scenario might appear in a streaming algorithm that only considers keys occurring in some time window or in a sweep-line algorithm on two dimensional data points (x_i, y_i) where the y coordinates of the points within a moving interval of the x axis are maintained in the data structure.

Finally, Proposition 12 gives space usage and initialization time of the index.

▶ **Definition 1.** *Given a table with linear Robin-Hood probing (from left to right) with wrap-around.*

1. *The function $h(x)$ maps a key x linearly to its slot.*
2. *The* displacement of a key *is the circular distance between the index of the slot it is placed and $h(x)$.*
3. *Let $d(x)$ be the displacement of key x. The* displacement of the slot i *is $|\{x \in S \mid i \in \{h(x), \dots, h(x) + d(x) - 1 \mod m\}\}|$, where S is the set of all keys in the table and m is the number of slots.*

More intuitively, if we insert all keys one after another then the displacement of a slot is the number of keys that probed the slot when it was already occupied. Hence, the displacement of the slot i is the circular distance to the last slot where a search for a key x with $h(x) = i$ might be found.

▶ **Lemma 2.** *In an RHT of m slots and $n \leq m$ keys, the sum of the displacements of the keys is equal to the sum of the displacements of the slots.*

Proof. For each key in the table $x \in S$, the set $P_x \in \{h(x), \dots, h(x) + d(x) - 1 \mod m\}$ is of size $d(x)$ and there are exactly $d(x)$ slots i such that $i \in P_x$. ◀

▶ **Lemma 3.** *Let S be a set of n random keys $X_1, \dots, X_n$, where each X_i is chosen according to a probability distribution on $[0, 1]$ with density function f_i (i.e. $X_i \sim f_i$). We define the highest probability across all distributions as $w_{\max} := \max_{i \in [n], x \in [0,1]} f_i(x)$. Let $\varepsilon > 0$ be a constant. Then for an RHT on S with $m \geq (1 + \varepsilon) w_{\max} n$ slots, the expected displacement of any slot is in $\mathrm{O}\left(\frac{1}{\varepsilon}\right)$.*

Proof. We generate $X_i \sim f_i$, using rejection sampling with proposal $Y_{i,j} \sim \mathcal{U}([0, 1])$ and acceptance probability $f_i(Y_{i,j})/w_{\max}$. Let T_i be the number of proposals required for X_i, and $T = \sum_{i=1}^{n} T_i$. Let $S' \supseteq S$ be the set of both accepted samples and rejected samples, i.e. $S' := \{Y_{i,j} : 1 \leq i \leq n, 1 \leq j \leq T_i\}$, so $|S'| = T$. We have $\mathbb{E}[T_i] = w_{\max}$, hence $\mathbb{E}[|S'|] = \mathbb{E}[T] = w_{\max} n$. We first bound T, which is the sum of n i.i.d. geometric random variables. According to [31, Theorem 2.1.], $\mathbb{P}[T \geq (1 + \frac{\varepsilon}{2}) w_{\max} n] \leq e^{-\Omega(n)}$. We can therefore use the naive upper bound n on the displacement of a key in the RHT on S in the event that $T \geq (1 + \frac{\varepsilon}{2}) w_{\max} n$.

Removing keys from an RHT can only decrease the displacements of the slots. We therefore use the expected displacements of the RHT on S' as an upper bound for the expected displacements of the RHT on S.

According to [26], the mean displacement of the keys D is in $\mathrm{O}\left(\frac{1}{1-\alpha}\right)$, where $\alpha = \frac{T}{m}$ is the load of the table. With $m \geq (1 + \varepsilon) w_{\max} n$ and $T < (1 + \frac{\varepsilon}{2}) w_{\max} n$, we have $D \in \mathrm{O}\left(\frac{1}{\varepsilon}\right)$. With Lemma 2 the same bound applies to the sum of the displacements of the slots. By rotational symmetry the displacements of all slots in the RHT on S' have the same distribution. Hence, the expected displacement of any slot in the RHT on S' is $\mathrm{O}\left(\frac{1}{\varepsilon}\right)$ and this upper bounds the displacement of any slot in the RHT on S. ◀

Now that we have an upper bound for the distance a query has to probe because of the clustering, we also need an upper bound for the distance a query has to probe because of gaps.

▶ **Definition 4.** *The* right (left) gap of a slot i *is the circular distance towards the left (right) to the next slot j where at least one key k of the RHT has $h(k) = j$ (i.e. slot j is guaranteed to be occupied). The gap in an empty RHT is the number of slots.*

Note that, perhaps somewhat counterintuitively, this definition of a gap does not use the actual placement of the keys but only their initial position $h(x)$.

▶ **Lemma 5.** *Let S be a set of $n \geq 1$ random keys $X_1, \ldots, X_n$, where each X_i is chosen according to a probability distribution on $[0, 1]$ with density function f_i (i.e. $X_i \sim f_i$). We define the lowest probability across all distributions as $w_{\min} = \min_{i\in[n],x\in[0,1]} f_i(x)$.*

For an RHT on S with $m \geq n$ slots, the expected left and right gap at any slot is $\mathrm{O}\left(\frac{m}{nw_{\min}}\right)$.

Proof. Let s be any slot, whose left and right gaps we are interested in. We use rejection sampling with n proposals $X_i \sim f_i$. We accept a sample with probability $\frac{w_{\min}}{f_i(X_i)}$ and if $h(X_i) \neq s$. The expected probability that a sample is accepted is $\int_0^1 \frac{w_{\min}}{f_i(t)} f_i(t)\ \mathrm{d}t = w_{\min}$. The samples that pass the first condition are uniformly distributed. Conditioned on passing the first condition, the second one is passed with probability $\frac{m-1}{m}$. Let S' be the set of all accepted samples. The expected gap of s in an RHT on S' with m slots is an upper bound for the expected gap of s in the RHT on S, because removing keys from S can only increase gaps.

We have that $|S'|$ is binomially distributed with expectation $\frac{nw_{\min}(m-1)}{m}$. Using a Chernoff bound, the event $|S'| \geq \frac{(1-\varepsilon)nw_{\min}(m-1)}{m}$ fails with probability $\mathrm{O}\left(\frac{1}{nw_{\min}}\right)$, for some constant $\varepsilon > 0$. If the event fails, then we use that the gap is at most m and the contribution to the resulting expectation is at most $\mathrm{O}\left(\frac{m}{nw_{\min}}\right)$. We therefore assume $|S'| \geq \frac{(1-\varepsilon)nw_{\min}(m-1)}{m}$ henceforth. For each slot j ($\neq s$), the probability p_{occupied} that at least one key $x \in S'$ has $h(x) = j$ is at least

$$\begin{aligned} p_{\text{occupied}} &= 1 - \left(1 - \frac{1}{m-1}\right)^{|S'|} = 1 - \left(\left(1 - \frac{1}{m-1}\right)^{m-1}\right)^{\frac{(1-\varepsilon)nw_{\min}}{m}} \\ &\geq 1 - e^{-\frac{(1-\varepsilon)nw_{\min}}{m}} \geq \left(1 - \frac{1}{e}\right)\frac{(1-\varepsilon)nw_{\min}}{m}. \end{aligned}$$

Correlations also work in our favor: if no key maps to a slot then another slot has a higher probability of being occupied. Hence, the expected gap of slot s is upper bounded by a geometric random variable with expectation $\frac{1}{p_{\text{occupied}}} \in \mathrm{O}\left(\frac{m}{nw_{\min}}\right)$. ◀

▶ **Lemma 6.** *Given $x \sim f = \mathcal{N}(\mu, \sigma^2)$ then $\mathbb{P}\left[\frac{\max_{y\in[x-\delta,x+\delta]} f(y)}{\min_{y\in[x-\delta,x+\delta]} f(y)} > \gamma\right] \leq 2\exp\left(-\frac{\sigma^2 \ln^2 \gamma}{32\delta^2}\right)$ where $\frac{\delta}{\sigma} \to 0$ and $\gamma > 1$.*

Proof. Using a tail bound on $\mathcal{N}$ we have for $\mu = 0$

$$\begin{aligned}
&\mathbb{P}\left[\frac{\max_{y\in[x-\delta,x+\delta]} f(y)}{\min_{y\in[x-\delta,x+\delta]} f(y)} > \gamma\right] = \mathbb{P}\left[\frac{f(|x|-\delta)}{f(|x|+\delta)} > \gamma\right] \\
&= \mathbb{P}\left[\frac{\exp(-(|x|-\delta)^2/\sigma^2)}{\exp(-(|x|+\delta)^2/\sigma^2)} > \gamma\right] = \mathbb{P}\left[\exp\left(\frac{4|x|\delta}{\sigma^2}\right) > \gamma\right] \\
&= \mathbb{P}\left[|x| > \frac{\sigma^2 \ln\gamma}{4\delta}\right] \leq 2\exp\left(-\frac{(\frac{\sigma^2\ln\gamma}{4\delta})^2}{2\sigma^2}\right) = 2\exp\left(-\frac{\sigma^2\ln^2\gamma}{32\delta^2}\right)
\end{aligned}$$

◀

In the following let $\mathcal{N}_{[0,1]}(\mu_i, \sigma^2)$ be a normal distribution truncated to $[0,1]$.

▶ **Lemma 7.** *Given n values $\mu_1, \ldots, \mu_n \in [0,1]$ as deterministic input. Our index is constructed with a bucket length of $\delta(n)$ on the n smoothed keys k_i, i.e. each k_i is chosen according to $\mathcal{N}_{[0,1]}(\mu_i, \sigma^2)$. Let σ be at least $\Omega(\delta\sqrt{\ln n})$. Then, excluding the time an operation spends for operating on the vEB (this is accounted for later) we have:*

- **(I)** *The expected time for an arbitrary successor query is constant.*
- **(II)** *The expected time for an arbitrary predecessor query is constant.*
- **(III)** *The expected time for an insertion of a key $k' \sim \mathcal{N}_{[0,1]}(\mu_i, \sigma^2)$ is amortized constant for any $\mu' \in [0,1]$.*
- **(IV)** *The expected time for a deletion of a key is amortized constant.*

Proof. We sample k_i in two steps. First we reveal the buckets that the keys map into. Then let f_i be the conditional distribution of key k_i within its bucket. We choose the constant γ such that $1 < \gamma < \frac{1}{(1+\varepsilon)\alpha_{\max}}$, where $\varepsilon > 0$ and a suitable $\alpha_{\max}$ that decides when the RHT grows. According to Lemma 6 the event $\frac{\max f_i(x)}{\min f_i(x)} \leq \gamma$ fails with probability $\exp(-\Theta(\frac{\sigma^2}{\delta^2})) \subseteq \mathrm{O}(\frac{1}{n})$, because $\sigma \in \Omega(\delta\sqrt{\ln n})$. If the unlikely event fails we apply the naive upper bound $\mathrm{O}(n)$ on all operations. We therefore assume $\frac{\max f_i(x)}{\min f_i(x)} \leq \gamma$. Then $\max f_i(x) \leq \gamma$ and $\min f_i(x) \geq \frac{1}{\gamma}$ because $\int f(x)dx = 1$. Therefore the conditions of Lemma 3 and Lemma 5 are fulfilled and we have that the expected displacement and left/right gap at any slot and in all buckets is constant.

- **(I)** A successor query for k begins in slot $h(k)$ of the corresponding bucket of k. Let d be the displacement and g the right gap of $h(k)$. Then the result of the query is found in the slots between $h(k)$ and $h(k)+d+1 \mod m$, if the result is within the current cluster. If this is not the case then the result is found in $h(k)+g \mod m$. In both cases only a constant number of slots are probed in expectation. If the successor was not found in the bucket because there exists no larger key in the current bucket, then the vEB is used to find the next non-empty successor bucket (if no such bucket exists then there is no successor key). The successor query is repeated on the first slot of that bucket, again taking constant time in expectation.
- **(II)** A predecessor query for k is similar. Let g be the left gap and d be the displacement of $h(k)$. If $h(k)$ is occupied then we are in a cluster and the result is found between $h(k)$ and $h(k)+d \mod m$. Otherwise, we probe towards the left and we will find a predecessor in slot $h(k)-g \mod m$ or earlier as any displacement brings the predecessor closer to $h(k)$. If no predecessor exists we proceed analogously to the successor query.
- **(III)** The sum of the displacements of the keys of a bucket is the time spent to insert all of them. The expected sum of the displacements is linear in the number of keys (Lemma 3). Hence, the expected time for an insertion is constant.

(IV) Analogously, the expected time of a deletion is the difference between the sum of the displacements before and after the deletion and therefore also constant. If an insertion increases the load of the RHT above $\alpha_{\max}$ or a deletion decreases the load below $\alpha_{\min}$ then the RHT is resized and rebuilt. Using a standard argument that involves the potential method, the rebuilding time is amortized constant. ◀

Unfortunately, in the worst case our operations have to operate on the vEB and are therefore dominated by the time complexity of the vEB.

▶ **Proposition 8.** *Query operations have expected running time* $\mathrm{O}\left(\log\log\frac{1}{\delta}\right)$. *The same holds for insertions and deletions in the amortized sense.*

Proof. All operations need to make at most one operation on the vEB. Queries have to make at least one query to the vEB to find the predecessor or successor bucket. Deletions have to delete the bucket from the vEB if they deleted the last key. Insertions have to insert the bucket into the vEB if they insert the first key. One operation on the vEB takes time $\mathrm{O}\left(\log\log\frac{1}{\delta}\right)$ because there are $\frac{1}{\delta}$ buckets managed by the vEB. Regarding the work within buckets we use Lemma 7. ◀

We can work around the vEB overhead by considering a *burst* of operations. A burst operation performs the *same* operation on a set of keys. Their algorithms are described in the following. In case of queries we first determine the predecessor or successor bucket of all buckets using a linear pass (ignoring the vEB). We can then answer all queries as usual without accessing the vEB. In case of insertions and deletions we only perform them within the RHTs and rebuild the vEB afterwards.

▶ **Theorem 9.** *If the input size for the burst is at least* $\Omega\left(\frac{1}{\delta}\right)$ *then the burst operation is expected constant time per input key (amortized in case of insertions and deletions).*

Proof. Immediate, using Lemma 7 and the linear construction time of a vEB. ◀

However, burst operations are not always suitable. The vEB overhead occurs on highly adversarial inputs, querying keys that are on the edges of the buckets, inserting and then deleting a key to an empty bucket and so on. In an attempt to more closely mimic real-world inputs we consider a model of a (slightly) weaker adversary. Within this model, we obtain constant time operations. We refer to operations within this model as *In-Distribution* operations. In the in-distribution model, the keys that an adversary may choose look similar to the already inserted data. More formally we have:

▶ **Theorem 10.** *Given* n *values* $\mu_1, \ldots, \mu_n \in [0,1]$ *as deterministic inputs, where* n *is at least* $\Omega\left(\frac{1}{\delta}\lg\lg\frac{1}{\delta}\right)$. *We sample a set of smoothed keys* $K = \{k_1, \ldots, k_n\}$ *where* k_i *is chosen according to* $\mathcal{N}_{[0,1]}(\mu_i, \sigma^2)$. *For a randomly chosen* $k \sim \mathcal{U}(K)$ *we have that*

(I) *querying* k *has expected constant time on an index built on* $K \setminus \{k\}$,
(II) *inserting* k *has expected amortized constant time on an index built on* $K \setminus \{k\}$ *and*
(III) *deleting* k *has expected amortized constant time on an index built on* K.

All other parameters are as in Lemma 7.

Proof. The time for all operations within the RHTs is expected (amortized) constant according to Lemma 7. We only have to show that operations on the vEB are sufficiently unlikely and disappear in the expectation for operations on the randomly chosen smoothed key k. Operations on the vEB have time $\mathrm{O}\left(\lg\lg\frac{1}{\delta}\right)$. Hence, if at most $\frac{1}{\delta}$ keys cause a vEB operation, we have expected constant time because $n \in \Omega\left(\frac{1}{\delta}\lg\lg\frac{1}{\delta}\right)$.

(I) In case of a predecessor (or successor) query on k, there can only be at most as many keys in K the lowest (or highest) key of a bucket as there are buckets. Only those keys will cause a vEB query when selected as k for an index on $K \setminus \{k\}$.

(II) In case of an insertion of k, consider the index built on K, then there can be at most as many keys alone in a bucket as there are buckets. If we sample one of those keys as k, then they will be the first key in their bucket in an index built on $K \setminus \{k\}$.

(III) Deletions are analogous, only the keys that are alone in a bucket cause a vEB deletion. ◀

▶ **Theorem 11.** *Given a capacity n and parameters as in Lemma 7, consider an adversary that specifies a sequence of insertion values $\mu_1, \mu_2, \ldots \in [0,1]$. We insert smoothed keys $k_1, k_2, \ldots$ one after another in this order, where each k_i is chosen according to $\mathcal{N}_{[0,1]}(\mu_i, \sigma^2)$. Let state t denote the index immediately after the insertion of k_t.*

Between insertions, i.e., while the index is in state t, the adversary may issue queries and deletions, subject to the following constraint: a key k_i may only be deleted in state t if $i < t - T$, i.e., each key must remain in the index for at least T steps before it becomes eligible for deletion. The adversary must use deletions to ensure that the capacity n of the index is never exceeded.

Then, for $T \in \Omega\left(\frac{1}{\delta} \lg\lg \frac{1}{\delta}\right)$, insertions and deletions have amortized constant expected time.

Hence, constant time insertions and deletions can also be achieved if keys stay in the index for at least T steps before deletion.

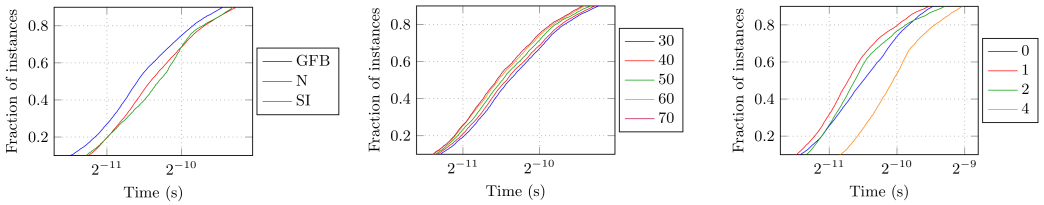
Proof. From the perspective of any bucket, after the first key is inserted into the bucket, the key will stay there for at least T steps. Hence, a bucket can only be inserted to the vEB at most every T steps. With the vEB insertion, we immediately charge twice the cost for operating on the vEB, making the possible vEB deletion at a later point free in the amortized sense. There are $\frac{1}{\delta}$ buckets causing a cost of $\mathrm{O}\left(\lg\lg \frac{1}{\delta}\right)$ at most every T steps. Hence, the amortized cost after at least $T \in \Omega\left(\frac{1}{\delta} \lg\lg \frac{1}{\delta}\right)$ steps is constant. The operations within the buckets are constant in expectation (Lemma 7). ◀

▶ **Proposition 12.** *PARROT is initialized in time $\mathrm{O}\left(\frac{1}{\delta}\right)$ and has space usage of $\mathrm{O}\left(N + \frac{1}{\delta}\right)$, where N is the number of keys currently in the index.*

Proof. The index is initialized by initializing an array of $\frac{1}{\delta}$ empty buckets and an empty vEB, where the buckets are its universe. Hence, the initialization takes time $\mathrm{O}\left(\frac{1}{\delta}\right)$. The space usage of the index is proportional to (I) the number of buckets and (II) the sum of the slots of all buckets, which is proportional to N. Hence, the total space usage is $\mathrm{O}\left(N + \frac{1}{\delta}\right)$. ◀

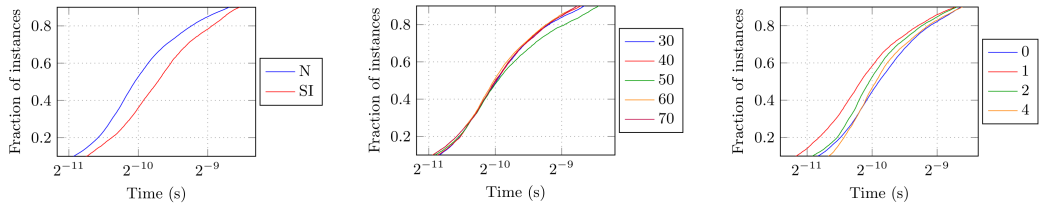
6 Experimental Analysis

In this section, we present a summary of the experimental results, while deferring additional figures and plots to Appendix C. All experiments are conducted on a machine equipped with Intel(R) Xeon(R) Gold 6238R CPU with frequency set to the nominal 2.20 GHz (32KB L1, 1MB L2, 39MB L3), 512 GB of RAM, Ubuntu 22.04 LTS on an x86-64 architecture, hyperthreading is disabled. The performance of PARROT and competitors is evaluated under multiple configurations using batches of $k = 10\,000$ operations, reporting execution time per operation. The experimented workloads are: (i) in-distribution lookups, where absent keys are sampled from the data distribution by uniformly selecting one of the $n + 1$ intervals



Bottom-level optimizations RHT load factor Unrolled SIMD probes

Figure 2 *Static-PARROT* Fraction of problem instances (y-axis) that complete within a given execution time (x-axis). The depicted bottom-level optimizations in subfigure (a) are: the optimization that fills the gap with the predecessor and successor (GFB), the bit vector at the RHT level (SI), or none optimization (N), the load factor in subfigure (b) is in percentage.



Bottom-level optimizations RHT load factor Unrolled SIMD probes

Figure 3 *Dynamic-PARROT* Fraction of problem instances (y-axis) that complete within a given execution time (x-axis).The depicted bottom-level optimizations in subfigure (a) are: the optimization that fills the gap with the predecessor and successor (GFB), the bit vector at the RHT level (SI), or none optimization (N), the load factor in subfigure (b) is in percentage.

between the sorted keys and then sampling uniformly within it [3]; (ii) uniform lookups, with keys drawn uniformly over the data range, independent of the distribution; (iii) insertions followed by deletions, where we start from the empty data structure, fully load it and then delete all the keys in the same order they were inserted.

PARROT is highly configurable: the number of unrolled SIMD probes, whether bit vectors are enabled, the use of gap filling, and the number of high bits. Our first step is therefore to perform a search for the best-performing configuration across four synthetic datasets, using 32-bit domains. Specifically, we consider the following distributions: uniform; normal ($\mu = \frac{1}{2}$, $\sigma = \frac{1}{4}$); a mixture of two narrow Gaussians that are well separated using the following means, standard deviations and weights ($\mu_1 = \frac{1}{4}$, $\sigma_1 = \frac{1}{50}$, $w_1 = 0.3$, $\mu_2 = \frac{3}{4}$, $\sigma_2 = \frac{1}{100}$, $w_2 = 0.7$); and the exponential distribution ($\lambda = 2$). All samples are scaled to the 32-bit domain and rounded to the nearest integer.

Figures 2 and 3, show the ablation study we conducted for tuning: the number of SIMD probes to be unrolled when probing the RHT, the RHT load factor, and the most effective optimization strategy at the bottom level. Although these three parameters depend on both the data distribution and the underlying hardware, we find that, in general, a robust choice is to adopt a configuration with one SIMD vector probed ahead, have a RHT load factor of 0.7, and apply the *gap-fill* optimization for static PARROT.

[3] We consider this more interesting then just querying a randomly chosen key exactly, where one could “cheat” using a hash table with precomputed results. We remark that this definition of in-distribution slightly deviates from the in-distribution model used in theory (see Theorem 10), where we delete a randomly chosen key and then query it.

After conducting the preliminary study in this controlled setting, we extend the evaluation to compare PARROT with well-established learned and traditional indexes that are relevant for being either the state of the art, or similar in their algorithmic structure to PARROT. We considered: Radix Spline (RS) [33], ALEX [19], LIPP [60], PGM [24], a well engineered non-learned B-Tree [9], and the predecessor data structure of [20], which we name after the authors' initials DFH. The results are shown in Figure 4, and additional pictures are in the Appendix C.

We observe that PARROT is the fastest on most of the depicted workloads and datasets. Moreover, we observe:

- For simplicity, all variants of PARROT depicted in Figure 4 use $b = 12$ bits. We observe that this choice is sufficient to partition the key space, spanning the full unsigned integer domain, at a granularity that yields stable and smooth performance, even under skewed distributions such as the mixture of Gaussians and the exponential distribution.
- Dynamic learned index structures such as ALEX, LIPP, and dynamic PGM are consistently outperformed across all evaluated workloads by most of the considered predecessor data structures.
- Classical non-learned data structures, such as B-Trees and Y-fast tries, are generally slower than learned index-based approaches across the evaluated workloads.
- The DFH data structures are very good large datasets. Perhaps, with further tuning, it could also be an excellent choice on smaller datasets.

Table 1 PARROT space occupancy.

Keys	Uniform	Normal	Mix Gauss	Exponential
2^{20}	8.23MB	9.35MB	8.06MB	8.50MB
2^{23}	62.95MB	69.27MB	63.09MB	63.89MB
2^{26}	501.97MB	548.56MB	503.59MB	506.39MB
2^{29}	4.00GB	4.29GB	4.01GB	4.03GB

Table 1 reports the memory footprint of PARROT, as the number of keys increases in the four datasets. Memory scales linearly with the number of keys. The behaviour is expected. Each RHT is sized for a target load factor of 0.7 and rounded up to the next power of two. For the uniform distribution, where every bucket receives approximately the same number of keys, the measured size closely matches the ideal. The normal distribution, in contrast, packs the keys around the mean, pushing some RHT to double their size to the next power-of-two. The heavily skewed distributions exhibit the opposite effect: many buckets in the sparsely populated tails end up nearly empty, thus remaining small.

We also consider two of the well-known 64-bits SOSD datasets [42]: Wikipedia article edit timestamps (without duplicates), and book sales popularity. Figure 5 illustrates the lookup workloads for both in-distribution and uniform queries across the two SOSD datasets. Here we set b in such a way that the number of non-empty bucket is 2^{16} which fits L2 cache. In this setting, the optimal configuration for PARROT differs due to the distinct characteristics of the 64-bits datasets and real world distributions, which favor a lower load factor and a longer probe length. Some indices are omitted from the figure: LIPP does not support searches for non-existent keys, and we observed that ALEX may produce errors on 64-bit keys, which are arguably due to machine precision issues in its internal computations.

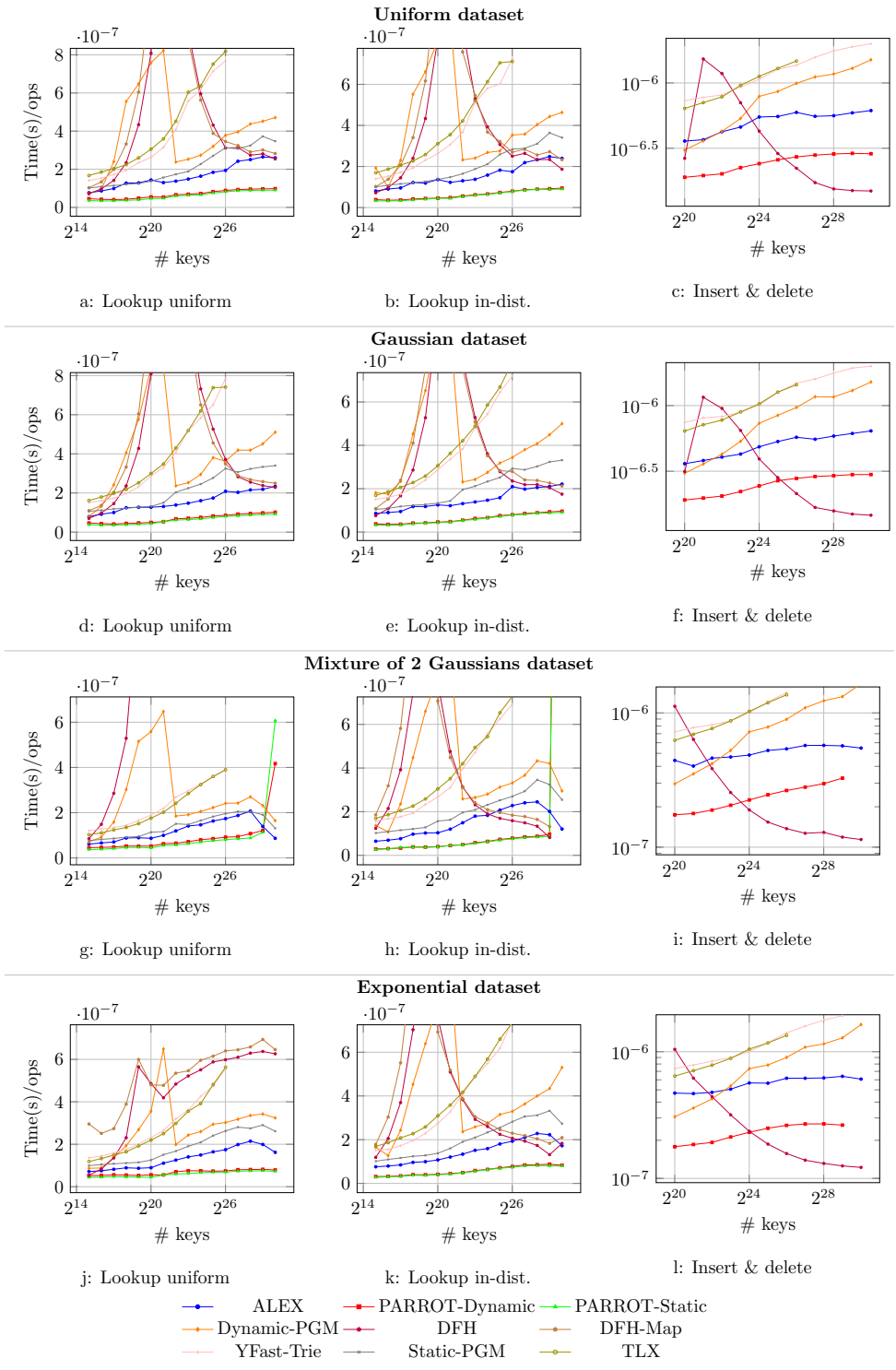


Figure 4 Execution time for a batch of 10000 operations over four 32-bits unsigned integer synthetic datasets. The fastest variant of each index is selected. Missing data series are out of range. Experiments for lookups of existing keys in C. For readability Insert-Delete workload is in log scale.

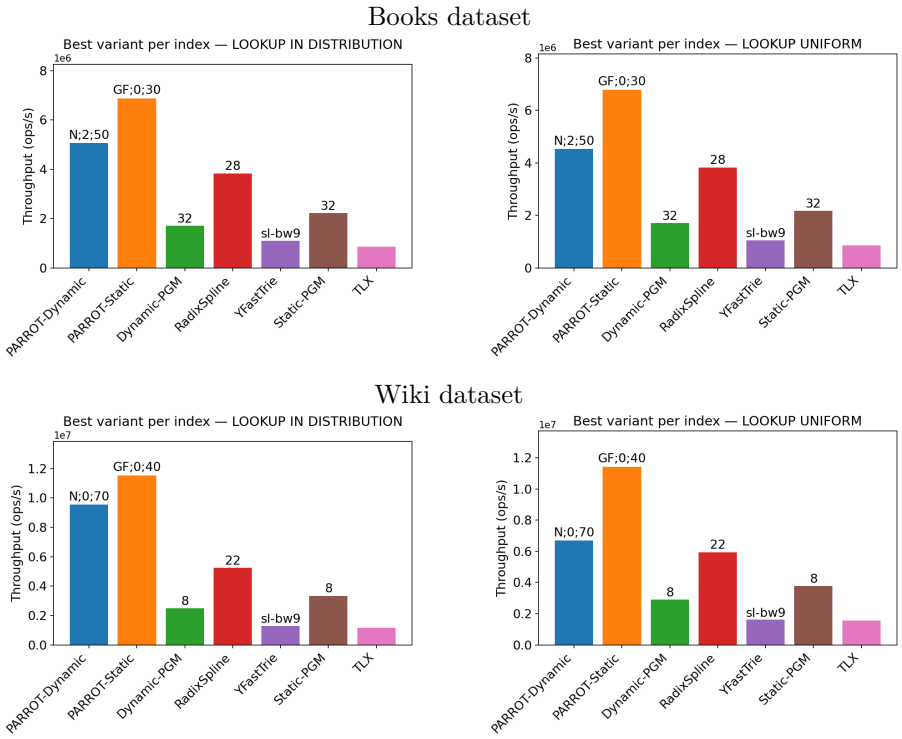


Figure 5 Throughput of in-distribution (left) and uniform (right) lookup queries on two real world SOSD datasets: books (top) and wiki (bottom). The best configuration for each index is shown on top of the bar. For PARROT these are of the form: gap-fill (GF) or none (N); number of unrolled SIMD probes; RHT load factor. For static and dynamic PGM-index it is the ε parameter, and for RadixSpline it is the number of high bits. All YFast-Tries stores the keys as sorted lists (sl).

7 Conclusion and Future Work

Smoothed analysis is a natural model for understanding (learned) index data structures and helps design various concrete data structures using a number of components like B-Tree nodes, arrays, and hash tables. Likely, this will also be interesting for other learned indexes. With PARROT, we show that minimal learning is enough to break down the input into approximately uniformly distributed pieces, resulting in good performance in many cases.

A lot of subsequent work suggests itself. In particular, we might use different ways to arrange building blocks, perhaps dynamically. For example, adding B-Tree nodes at the top or the bottom (see Appendix B.1 for details). More levels of arrays/hash tables could zoom in on more narrow regions of a distribution. We would get a space efficient learned index by storing a sample of keys in a PARROT-like data structure that has references into a sorted array as payload. For example, by sampling every $\log n$-th key, and using interpolation search, we may get a succinct index with $O(\log \log \log n)$ query time on smoothed inputs. See Appendix B.2 for an even more sophisticated approach. Also specializing for particular usage patterns makes sense. See Appendix B.3 for an outline on monotone priority queues.

We used Gaussians for modeling noise mainly out of tradition. We expect that our analysis can be adapted to various other distributions. Perhaps more interestingly, we could look at a model where measurement errors are not additive but multiplicative which seems

more natural when keys span a wide range. It seems that this could be handled on the data structure side by a logarithmic transformation of the keys which converts multiplicative noise to additive noise. By approximating the logarithm[4] this would likely be quite efficient.

Parallelizing PARROT seems promising because the bottom level resembles linear probing hash tables which can work concurrently [41] and because generally the structure is less fluid than balanced search trees.

References

1 Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G. Aref. A survey of learned indexes for the multi-dimensional space, 2024. doi:10.48550/arXiv.2403.06456.

2 Abdullah Al-Mamun, Hao Wu, Qiyang He, Jianguo Wang, and Walid G. Aref. A survey of learned indexes for the multi-dimensional space. *ACM Comput. Surv.*, 58(4), October 2025. doi:10.1145/3768575.

3 O. Amble and D. E. Knuth. Ordered hash tables. *The Computer Journal*, 17(2):135–142, January 1974. doi:10.1093/comjnl/17.2.135.

4 Arne Andersson and Christer Mattsson. Dynamic interpolation search in o (log log n) time. In *International Colloquium on Automata, Languages, and Programming*, pages 15–27. Springer, 1993. doi:10.1007/3-540-56939-1_58.

5 R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972. doi:10.1007/BF00288683.

6 R. Beier and B. Vöcking. Random knapsack in expected polynomial time. In *35th ACM Symposium on Theory of Computing*, 2003.

7 Djamal Belazzougui, Alexis C. Kaporis, and Paul G. Spirakis. Random input helps searching predecessors. In Luca Ferrari and Malvina Vamvakari, editors, *Proceedings of the 11th International Conference on Random and Exhaustive Generation of Combinatorial Structures, GASCom 2018, Athens, Greece, June 18-20, 2018*, volume 2113 of *CEUR Workshop Proceedings*, pages 106–115. CEUR-WS.org, 2018. URL: https://ceur-ws.org/Vol-2113/paper10.pdf.

8 Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $o(nlogn)$. *Theor. Comp. Sys.*, 39(3):391–397, June 2006. doi:10.1007/s00224-005-1237-z.

9 Timo Bingmann. TLX: Collection of sophisticated C++ data structures, algorithms, and miscellaneous helpers, 2018. , retrieved Apr. 20, 2026. URL: https://panthema.net/tlx.

10 Antonio Boffa, Paolo Ferragina, and Giorgio Vinciguerra. A learned approach to design compressed rank/select data structures. *ACM Transactions on Algorithms*, 2022. doi:10.1145/3524060.

11 Randy Brown. Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, 1988. doi:10.1145/63039.63045.

12 Pedro Celis, Per-Ake Larson, and J. Ian Munro. Robin hood hashing. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 281–288, USA, 1985. IEEE Computer Society. doi:10.1109/SFCS.1985.48.

13 Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. Airindex: Versatile index tuning through data and storage. *Proc. ACM Manag. Data*, 1(3):204:1–204:26, 2023. doi:10.1145/3617308.

14 Luis Alberto Croquevielle, Roman Sokolovskii, and Thomas Heinis. Lower bounds for the algorithmic complexity of learned indexes, 2026. doi:10.48550/arXiv.2601.06629.

15 Luis Alberto Croquevielle, Guang Yang, Liang Liang, Ali Hadian, and Thomas Heinis. Beyond Logarithmic Bounds: Querying in Constant Expected Time with Learned Indexes. In *28th International Conference on Database Theory (ICDT 2025)*, volume 328, pages 19:1–19:21, 2025. doi:10.4230/LIPIcs.ICDT.2025.19.

[4] For example, we could add a layer at the top that differentiates keys by their most significant 1-bit.

16 Andrew Crotty. Hist-tree: Those who ignore it are doomed to learn. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org, 2021. URL: https://vldb.org/cidrdb/2021/hist-tree-those-who-ignore-it-are-doomed-to-learn.html.

17 Erik D Demaine, Thouis Jones, and Mihai Pătraşcu. Interpolation search for non-independent data. In *15th ACM-SIAM symposium on Discrete algorithms (SODA)*, pages 529–530, 2004.

18 R. Dementiev, L. Kettner, J. Mehnert, and P. Sanders. Engineering a sorted list data structure for 32 bit keys. In *6th Workshop on Algorithm Engineering & Experiments*, pages 142–151, New Orleans, 2004.

19 Jialin Ding, Umar Farooq Minhas, Jia Yu, Chi Wang, Jaeyoung Do, Yinan Li, Hantian Zhang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, David Lomet, and Tim Kraska. ALEX: An updatable adaptive learned index. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, pages 969–984, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3318464.3389711.

20 Patrick Dinklage, Johannes Fischer, and Alexander Herlez. Engineering Predecessor Data Structures for Dynamic Integer Sets. In *19th International Symposium on Experimental Algorithms (SEA 2021)*, volume 190, pages 7:1–7:19, 2021. doi:10.4230/LIPIcs.SEA.2021.7.

21 Paolo Ferragina and Filippo Lari. Compressibility Measures and Succinct Data Structures for Piecewise Linear Approximations. In *36th International Symposium on Algorithms and Computation (ISAAC 2025)*, volume 359 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:15, 2025. doi:10.4230/LIPIcs.ISAAC.2025.31.

22 Paolo Ferragina and Filippo Lari. FL-RMQ: A Learned Approach to Range Minimum Queries. In *36th Annual Symposium on Combinatorial Pattern Matching (CPM 2025)*, volume 331, pages 7:1–7:23, 2025. doi:10.4230/LIPIcs.CPM.2025.7.

23 Paolo Ferragina, Hans-Peter Lehmann, Peter Sanders, and Giorgio Vinciguerra. Learned monotone minimal perfect hashing. In *31st European Symposium on Algorithms (ESA)*, 2023. doi:10.4230/LIPIcs.ESA.2023.46.

24 Paolo Ferragina, Fabrizio Lillo, and Giorgio Vinciguerra. On the performance of learned data structures. *Theoretical Computer Science*, 871:107–120, 2021. doi:10.1016/j.tcs.2021.04.015.

25 Paolo Ferragina and Giorgio Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020. doi:10.14778/3389133.3389135.

26 Philippe Flajolet, Patricio Poblete, and Alfredo Viola. On the analysis of linear probing hashing. *Algorithmica*, 22(4):490–515, 1998. doi:10.1007/PL00009236.

27 Greg N. Frederickson. Implicit data structures for the dictionary problem. *Journal of the ACM (JACM)*, 30(1):80–94, 1983. doi:10.1145/322358.322364.

28 Emil Toftegaard Gæde, Ivor van der Hoog, Eva Rotenberg, and Tord Stordalen. A Dynamic Piecewise-Linear Geometric Index with Worst-Case Guarantees. In *33rd Annual European Symposium on Algorithms (ESA 2025)*, volume 351, pages 64:1–64:18, 2025. doi:10.4230/LIPIcs.ESA.2025.64.

29 Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. Fiting-tree: A data-aware index structure. In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 1189–1206, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3299869.3319860.

30 Alon Itai, Alan G. Konheim, and Michael Rodeh. A sparse table implementation of priority queues. In Shimon Even and Oded Kariv, editors, *Automata, Languages and Programming, 8th Colloquium, Acre (Akko), Israel, July 13-17, 1981, Proceedings*, volume 115 of *Lecture Notes in Computer Science*, pages 417–431. Springer, 1981. doi:10.1007/3-540-10843-2_34.

31 Svante Janson. Tail bounds for sums of geometric and exponential variables. *Statistics & Probability Letters*, 135:1–6, 2018.

32 Alexis C. Kaporis, Christos Makris, Spyros Sioutas, Athanasios K. Tsakalidis, Kostas Tsichlas, and Christos D. Zaroliagis. Dynamic interpolation search revisited. *Inf. Comput.*, 270, 2020. doi:10.1016/J.IC.2019.104465.

33 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*, aiDM '20, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3401071.3401659.

34 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*, SIGMOD '18, pages 489–504, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3183713.3196909.

35 Ani Kristo, Kapil Vaidya, and Tim Kraska. Defeating duplicates: A re-design of the learnedsort algorithm. *CoRR*, abs/2107.03290, 2021. arXiv:2107.03290.

36 Pengfei Li, Yu Hua, Jingnan Jia, and Pengfei Zuo. Finedex: a fine-grained learned index scheme for scalable and concurrent memory systems. *Proc. VLDB Endow.*, 15(2):321–334, October 2021. doi:10.14778/3489496.3489512.

37 Pengfei Li, Hua Lu, Rong Zhu, Bolin Ding, Long Yang, and Gang Pan. Dili: A distribution-driven learned index. *Proc. VLDB Endow.*, 16(9):2212–2224, May 2023. doi:10.14778/3598581.3598593.

38 Liang Liang, Guang Yang, Ali Hadian, Luis Alberto Croquevielle, and Thomas Heinis. Swix: A memory-efficient sliding window learned index. *Proc. ACM Manag. Data*, 2(1), March 2024. doi:10.1145/3639296.

39 Qiyu Liu, Siyuan Han, Yanlin Qi, Jingshu Peng, Jin Li, Longlong Lin, and Lei Chen. Why are learned indexes so effective but sometimes ineffective? *Proc. VLDB Endow.*, 18(9):2886–2898, September 2025. doi:10.14778/3746405.3746415.

40 Chaohong Ma, Xiaohui Yu, Yifan Li, Aishan Maoliniyazi, and Xiaofeng Meng. LINDAS: a learned approach to index algorithm selection. *Knowl. Inf. Syst.*, 67(12):11381–11423, 2025. doi:10.1007/S10115-025-02552-W.

41 Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent hash tables: Fast and general?(!). *ACM Transactions on Parallel Computing (TOPC)*, 5, 2019. doi:10.1145/3309206.

42 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, September 2020. doi:10.14778/3421424.3421425.

43 Kurt Mehlhorn and Stefan Näher. Bounded ordered dictionaries in $O(\log \log N)$ time and $O(n)$ space. *Information Processing Letters*, 35(4):183–189, 1990. doi:10.1016/0020-0190(90)90022-P.

44 Kurt Mehlhorn and Athanasios Tsakalidis. Dynamic interpolation search. *J. ACM*, 40(3):621–634, July 1993. doi:10.1145/174130.174139.

45 Gonzalo Navarro and Javiel Rojas-Ledesma. Predecessor search. *ACM Comput. Surv.*, 53(5), September 2020. doi:10.1145/3409371.

46 W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957. doi:10.1147/rd.12.0130.

47 Mihai Pătraşcu and Mikkel Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '06, pages 232–240, New York, NY, USA, 2006. Association for Computing Machinery. doi:10.1145/1132516.1132551.

48 Atsuki Sato and Yusuke Matsui. PCF learned sort: a learning augmented sort algorithm with $\mathcal{O}(n \log \log n)$ expected complexity. *Transactions on Machine Learning Research*, 2025. Featured Certification, J2C Certification. URL: https://openreview.net/forum?id=wVkb8WHbvR.

49 J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *Journal of the ACM*, 10(2):217–255, 1963. doi:10.1145/321160.321170.

50 Benjamin Spector, Andreas Kipf, Kapil Vaidya, Chi Wang, Umar Farooq Minhas, and Tim Kraska. Bounding the last mile: Efficient learned string indexing. *CoRR*, abs/2111.14905, 2021. arXiv:2111.14905.

51 D. Spielman and S.-H. Teng. Smoothed analysis of algorithms: why the simplex algorithm usually takes polynomial time. In *33rd ACM Symposium on Theory of Computing*, pages 296–305, 2001. doi:10.1145/380752.380813.

52 Mihail Stoian, Andreas Kipf, Ryan Marcus, and Tim Kraska. PLEX: towards practical learned indexing. *CoRR*, abs/2108.05117, 2021. arXiv:2108.05117.

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

54 Chuzhe Tang, Youyun Wang, Zhiyuan Dong, Gansen Hu, Zhaoguo Wang, Minjie Wang, and Haibo Chen. Xindex: a scalable learned index for multicore data storage. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, pages 308–320, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3332466.3374547.

55 Kapil Vaidya, Subarna Chatterjee, Eric Knorr, Michael Mitzenmacher, Stratos Idreos, and Tim Kraska. Snarf: a learning-enhanced range filter. *Proc. VLDB Endow.*, 15(8):1632–1644, April 2022. doi:10.14778/3529337.3529347.

56 Peter van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inf. Process. Lett.*, 6(3):80–82, 1977. doi:10.1016/0020-0190(77)90031-X.

57 Youyun Wang, Chuzhe Tang, Zhaoguo Wang, and Haibo Chen. Sindex: a scalable learned index for string keys. In *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '20, pages 17–24, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3409963.3410496.

58 Dan E. Willard. Searching unindexed and nonuniformly generated files in log log n time. *SIAM Journal on Computing*, 14(4):1013–1029, 1985. doi:10.1137/0214071.

59 Dan E. Willard. Examining computational geometry, van emde boas trees, and hashing from the perspective of the fusion tree. *SIAM Journal on Computing*, 29(3):1030–1049, 2000. doi:10.1137/S0097539797322425.

60 Jiacheng Wu, Yong Zhang, Shimin Chen, Jin Wang, Yu Chen, and Chunxiao Xing. Updatable learned index with precise positions. *Proc. VLDB Endow.*, 14(8):1276–1288, April 2021. doi:10.14778/3457390.3457393.

61 Guang Yang, Liang Liang, Ali Hadian, and Thomas Heinis. FLIRT: A fast learned index for rolling time frames. In *EDBT*, pages 234–246, 2023. doi:10.48786/edbt.2023.19.

62 Yifan Yang and Shimin Chen. Lits: An optimized learned index for strings. *Proc. VLDB Endow.*, 17(11):3415–3427, July 2024. doi:10.14778/3681954.3682010.

63 Andrew C. Yao and F. Frances Yao. The complexity of searching an ordered random table. In *17th Annual Symposium on Foundations of Computer Science (sfcs 1976)*, pages 173–177, 1976. doi:10.1109/SFCS.1976.32.

64 Sepanta Zeighami and Cyrus Shahabi. On distribution dependent sub-logarithmic query time of learned indexing. In *International Conference on Machine Learning (ICML)*, pages 40669–40680. PMLR, 2023. URL: https://proceedings.mlr.press/v202/zeighami23a.html.

65 Sepanta Zeighami and Cyrus Shahabi. Theoretical analysis of learned database operations under distribution shift through distribution learnability. In *Proceedings of the 41st International Conference on Machine Learning*, ICML'24. JMLR.org, 2024.

A More Related Work

Given a sorted sequence S of n keys and a query q, we address the predecessor problem, i.e. find the largest $r \in S$ such that $r \leq q$. The standard approach relies on binary search, which answers queries in $O(\log n)$ time. However, this cost can be significantly reduced by augmenting S with an index, i.e. a data structure that is precomputed once and reused to accelerate query processing. This problem has been extensively studied: the literature offers dozens of indexing data structures. The lower bounds are also well understood; the static predecessor problem was solved optimally by Pătraşcu and Thorup [47]. Yet, these lower bounds do not apply when the index leverages information about the distribution of S.

Recently such indexes were rediscovered under the name *learned data structures* or *learned indexes*. However, the underlying idea of exploiting the data distribution is far older than the terminology suggests. Section A.3 surveys prior literature on such indexes that effectively employed forms of "learning" long before the term entered the mainstream. Importantly, many of these earlier approaches also provided rigorous and provable performance bounds. In the next Sections A.1 and A.2 we review the theoretical and practical aspects of learned indexes, respectively.

A.1 Learned indexes theory

A *learned index* is a data structure that approximates the Empirical Cumulative Distribution Function of S efficiently in time and space. It enables computing the *rank* of a key x, i.e. the number of keys in S less than or equal to x. Efficient rank computation underpins key primitives in large-scale data systems, including *membership*, *lookup*, *insert*, *delete*, and *predecessor*/*successor* queries. In this work, we do not directly address the problem of the rank computation, but we propose an index that support all the above mentioned primitives.

The proposal of the first learned index RMI [34] in 2018, started a novel and rapidly growing research field. Numerous works aimed to improve different aspects such as bounding the approximation error [25], dynamic updates [19, 60, 54, 61], string keys [62, 50, 57], and multiple dimensions [1]. More recent trends include the combination of learned and classic data structure [13], and the use of machine learning to select the "best" index [40]. At the same time, these data structures were finding several applications besides indexing: compressed *rank/select* indexing [10, 21], range filtering [55], hashing [23], sorting [35, 48], and range minimum queries [22], just to mention a few. Although most of the work on learned indexes emphasizes empirical performance in synthetic and real-world datasets [42, 53], relatively few articles offer rigorous analytical guarantees.

Most of the learned indexes are hierarchical structures, where at each level a model is used to reduce the search space for the next level. The last level of the hierarchy stores the actual keys. The first attempt to establish a theoretical framework explaining the empirically observed strong performance is due to Ferragina et al. [24]. They show that the PGM-Index [25] achieves a query I/O complexity that is provably no worse than that of B-Trees. This result was later extended to the RAM model in [39], where it was shown that, in expectation, the height of the PGM-Index, and hence its query time, is $O(\log \log n)$.

Zeighami et al. [64] show that, under some assumptions on the data distribution, the query time is sub-logarithmic on expectation over the data distribution. They present three indexes that stem from different trade-offs and assumptions on the density of the data. In particular, [64, Theorem 3.3] states that by assuming a bounded pdf $f_\chi(x)$, i.e., $\rho_1 \leq f_\chi(x) \leq \rho_2$ for all $x \in S$, where $\rho_1 > 0$ and $\rho_2 < \infty$, there exists a learned index with $O(\log \log n)$ expected query time, and $O(\frac{\rho_2}{\rho_1} n \log n)$[5] space; $\frac{\rho_2}{\rho_1}$ depends only on the distribution.

[5] As in [64], the $\log n$ bits is for storing the content of the node

The same authors [65] extended the theoretical framework to the dynamic case. The learned index is trained and data distribution can change due to subsequent updates. The proposed learned index have query and insertion cost of $O(\log\log n + \log(n\delta))$, being δ the distribution shift, namely a value in $[0, 1]$ indicating how the distribution changes when new keys are inserted or removed. This bound tells us that learned indexes are exponentially faster than standard data structures if their distribution is well captured, while they have the same logarithmic performance when the distribution is completely shifted, i.e. $\delta = 1$.

Croquevielle et al. [15] further extend the field by achieving constant query time and linear space. The index uses a piecewise constant approximation, which partitions the universe of keys into $O(n)$ intervals and assigns the median rank to all keys within that interval. Croquevielle et al. [14] establish lower bounds for the query time of learned indexes.

A.2 Learned indexes in practice

A complete review of learned indexes is beyond the scope of this paper; we refer the reader to [1] for a comprehensive overview. We briefly discuss the related works most relevant to our setting, with particular emphasis on dynamic updates and provable guarantees. Learned indexes are commonly organized as hierarchical model structures. The seminal proposal of Kraska et al. [34] views an index as a DAG of primitive models (e.g., linear functions, piecewise functions, neural networks). Following this work, numerous improvements have been proposed by combining learned models with classical indexing techniques. FITing-Tree [29] adopts piecewise linear approximation (PLA) models integrated with B^+-Trees. The PGM-Index [25] employs PLA models throughout the entire hierarchy, providing worst-case bounds on the search error together with linear-time construction. Radix-Spline [33] is a two-level index consisting of a radix table and spline models; also extended to strings [50].

With the exception of the PGM-Index, these approaches do not support dynamic updates. The PGM-Index adopts the logarithmic method, i.e., it maintains a set of indexes of geometrically increasing sizes that are periodically reconstructed[6]. Subsequent research has therefore explored alternative mechanisms to support mutability. ALEX [19] represents leaf nodes as gapped arrays, where positions are predicted via linear models and conflicts are resolved through shifts. When the density of a node violates predefined thresholds, the node is split or merged according to a cost model that estimates the amortized cost of future operations. LIPP [60] stores keys directly within internal nodes and relies exclusively on linear models for navigation. Collisions trigger node expansions; as a result, lookups follow model predictions without additional in-node search. ALEX and LIPP use two orthogonal approaches in handling collisions. Many combinations, variants and integration with classical indexes, and applications were studied, we mention just a few of them. PLEX [52] integrates the learned models with the Hist-Tree [16]. FLIRT [61] and SWIX [38] target sliding-window workloads, where new data are only appended and old data "expire" from the window after a certain period of time. XIndex [54] and FINEdex [36] support efficient concurrent and non-blocking updates. DILI [37] combines the accuracy typical of bottom-up constructions with the efficiency of top-down layouts. Bottom-up indexes capture the empirical distribution more faithfully, at the price of irregular fanouts; top-down schemes, instead, partition the key domain uniformly, which improves traversal speed.

[6] A recent work [28] effectively breaks and merges PLA segments in place of periodic reconstructions.

A.3 Interpolation search

As also observed in [65] learned indexes are closely connected to *interpolation search* and related data structures. We start by reviewing here historical results, then we move our focus to solutions achieving constant time. We remark that these works significantly predate the "learned" terminology. Let's start with the classic example. When we manually search for a word in the English dictionary, for example the word "book", we will probably start somewhere around the beginning of the dictionary, where we expect to find words that start with the letter B. Also the size of the jumps from one page to the other is driven by what we expect to be the distance between the current page and the searched word. That is what interpolation search does. Firstly proposed by Peterson in 1957 [46], interpolation search was shown in [63] to take $\Theta(\log\log n)$ time if S has been drawn from a uniformly distributed random variable. Frederickson [27] provided an implicit dynamic data structure, that also supported insertion and deletions in $O(n^{\varepsilon})$, $\varepsilon > 0$ time and queries in $O(\log\log n)$ expected time. Itai et al. [30] improved the insertion and deletion times devising another data structure that takes $O(\log^2 n)$ time amortized. Willard [58] devised a variant of interpolation search for which the $\Theta(\log\log n)$ query time applies also to a class of distributions other than the uniform one. He named this class of distributions *μ-random*. Mehlhorn and Tsakalidis [44] further extended this result to a wider class of distributions, called *smooth distributions*, and proposed a dynamic data structure: the *interpolation search tree* (IST). The IST is a multiway tree where the degree of a node is (in the ideal case) the square root of the keys it covers. Thus, the number of keys covered by a node geometrically decreases with the node depth (like van Emde Boas Trees [56], and unsurprisingly, PGM-Index [39]), and the tree depth is $\Theta(\log\log n)$. They introduced the concept of an (f_1, f_2)-*smooth* probability density to regulate how keys are distributed within subintervals of the distribution. Informally, a distribution defined over an interval I is considered smooth if, for any of the $f_1(n)$ equal-length subintervals of I, the probability density remains below an upper bound function of $f_2(n)$ [32]. IST supports updates that may unbalance the tree. If the density is smooth, periodical rebuild guarantees $\Theta(\log n)$ amortized update time, and $\Theta(\log\log n)$ expected amortized query time. Query time is $\Theta(\log^2 n)$ and update time is $\Theta(n)$ in the worst case.

Andersson and Mattsson [4] observed that the data stored in IST nodes are not only approximating the inverse distribution function, but are estimating the sample density, which is more powerful. In modern terms, we would say that we should overfit the density approximation as much as possible. Their data structure, removes the hierarchy of the IST and replace it with a single level, plus an additional level of binary search trees to accommodate for future insertions. They proved that, $\Theta(\log\log n)$ query time holds for a class of smooth densities wider than [44], and they achieved $o(\log\log n)$ for some distribution.

Demaine et al. [17] generalize interpolation search to non-random input data. Their dynamic data structure allows all operations in $O(\log\Delta)$ *worst-case* time, Δ being the max gap ratio between consecutive keys. There is no requirement on the data being independently drawn. In the case of the uniform distribution, by the Chernoff bound $\Delta = O(\log^2 n)$ with high probability. Insertions and deletions are supported in $O(\log\Delta)$ amortized time.

Kaporis et al. [32] observed that all the previous bounds do not apply in the presence of duplicates. They propose a data structure with $O(\log\log n)$ expected query time irrespectively of the distinctness or not of the keys in the input sequence. It enlarges the class of smooth distributions for which $O(1)$ query time holds, and is sufficiently robust to maintain $O(\log\log n)$ query time w.h.p. for power law and binomial distributions.

Belazzougui et al. [7] focused on the distribution of the queries, and presented a data structure that answers predecessor queries in O(1) time w.h.p. if the queries are drawn from the uniform distribution. The data structure uses O(n) space. They also show that a two layer data structure formed by a vEB tree on top and q*-heaps [59] on the bottom supports predecessor queries in O(1) time w.h.p. using $O(n^{1+\delta})\ \forall \delta > 0$ space (i.e., superlinear).

B Reconstructing Additional Data Structures

B.1 Using B-Tree Nodes

We already noted that using a B-Tree node at the *top* can split off special values like ∞. The overhead over basic PARROT is likely to be small as this root nodes will likely reside in L1-cache and because branch prediction has a good chance of catching the most frequent case. When the input is not sufficiently uniform at the bottom level, our RHTs will degenerate. This can be made more robust by using B-Trees to store our bottom-level buckets. This can also be engineered for low overhead compared to basic PARROT in benign regions of the sorted sequence. One cache line of our bottom-level array will store the root of a small B-Tree. A degree field will be zero when all keys fit into that cache line. In this case, there is no need to store pointers to children, and the remainder of the cache line can store keys. These can then be searched using SIMD instructions. Thus, the bottom level will locally behave like a bucketed hash table. These are known to perform even better in some cases than RHTs since everything is cache aligned. A nice theoretical side-effect of this measure is that the asymptotic search time goes from linear in the number of colliding keys to logarithmic.

B.2 Succinct Indexes and Monotone Minimal Perfect Hashing

A Monotone Minimal Perfect Hash Function (MMPHF) maps keys in a set S to their rank, i.e., it can be used as an index into a sorted array supporting the find operation but not successor/predecessor. LeMonHash [23] does this using a PGM index to partition the input into nearly linear pieces, an Elias-Fano encoded sequence of starting ranks for buckets of expected size 1, and a retrieval data structure for storing local ranks of buckets containing multiple keys. This takes 2.92 bits per key for uniformly distributed inputs. It seems that we can modify PARROT and our smoothed analysis to show that this also works for arbitrary smoothed inputs with sufficiently large σ while supporting constant time queries: The top-level table (without bit vectors) maps keys to top-level buckets. For sufficiently large σ, these are approximately uniformly distributed. Top-level buckets represent arrays of bottom-level buckets using an Elias-Fano coding and retrieval as in LeMonHash.
It seems that this can be further developed into a static index by ensuring that the data structure provides accurate bucket boundaries. For smoothed data, the expected bucket size is constant, yielding a constant expected query time. We can further reduce space by dropping the retrieval data structure and increasing the expected bucket size.

B.3 Monotone Priority Queues

We now look at a special case of a sorted sequence that only supports operations insert and deleteMin. Moreover, inserted keys may not exceed the most recently deleted key. This usage occurs in many applications of priority queues, e.g., Dijkstra's shortest path algorithm, best first branch-and-bound, or discrete event simulation.

We can simplify (and refine) PARROT to take advantage of this specialized usage pattern. We use no bitvector, vEB, or no hash table for the top level. Only a cyclic array of buckets (assuming that the range of keys concurrently in the queue has bounded size[7]). Only the logically leftmost nonempty bucket is expanded into a RHT. All other buckets are simply unsorted buffers. Now, insertions run in worst case constant time unless the leftmost bucket is addressed. There, we get constant expected time in an amortized sense. Here we can adopt the analysis from PARROT. DeleteMin is more complicated as we have no data structure accelerating skipping of empty buckets. However, this is no problem in an amortized sense as we can amortize skipping cost over the majority of deleteMin operations that work within the current minimal bucket. Optionally, we can support decreaseKey at the price of an additional indirection, e.g., using doubly-liked lists to represent (sub)buckets. This can be viewed as a more robust 2-level variant of the widely used calendar queue [11].

C More Experiments

We report two additional figures. Figure 6 depicts the lookup time for querying existing keys of the same datasets described in Section 6. Figure 7 depicts four workloads in a small dataset of geographic points. The keys are x coordinates, inserted in sorted order by y.

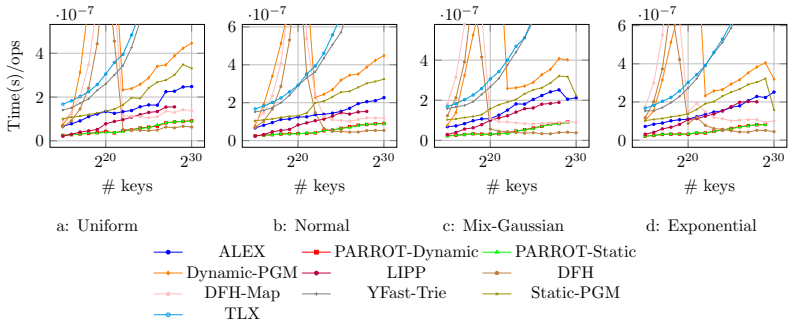


Figure 6 Lookup times of existing keys for the same data distributions depicted in figure 4. Here we also report LIPP which supports lookups of existing keys only.

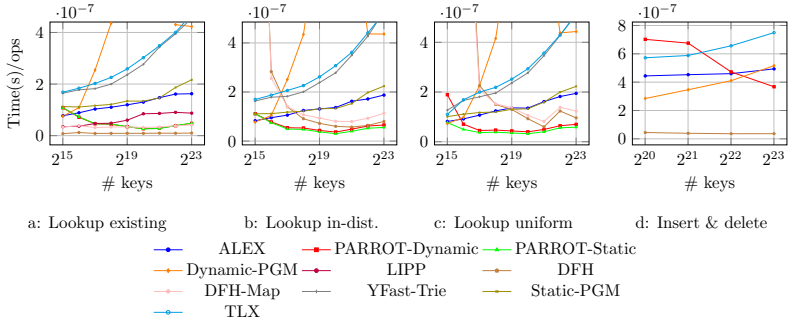


Figure 7 USA road network (uint32). Dataset of road networks of the US. The keys are the x coordinates of points, and are inserted in sorted order by y coordinate.

[7] This is the case for Dijkstra’s algorithm where the range of keys is bounded by the max edge weight.