The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds

Paolo Ferragina
University of Pisa, Italy
paolo.ferragina@unipi.it

Giorgio Vinciguerra
University of Pisa, Italy
giorgio.vinciguerra@phd.unipi.it

ABSTRACT

We present the first learned index that supports predecessor, range queries and updates within provably efficient time and space bounds in the worst case. In the (static) context of just predecessor and range queries these bounds turn out to be optimal. We call this learned index the Piecewise Geometric Model index (PGM-INDEX). Its flexible design allows us to introduce three variants which are novel in the context of learned data structures. The first variant of the PGM-INDEX is able to adapt itself to the distribution of the query operations, thus resulting in the first known *distribution-aware* learned index to date. The second variant exploits the repetitiveness possibly present at the level of the learned models that compose the PGM-INDEX to further *compress* its succinct space footprint. The third one is a *multicriteria* variant of the PGM-INDEX that efficiently auto-tunes itself in a few seconds over hundreds of millions of keys to satisfy space-time constraints which evolve over time across users, devices and applications.

These theoretical achievements are supported by a large set of experimental results on known datasets which show that the fully-dynamic PGM-INDEX improves the space occupancy of existing traditional and learned indexes by up to three orders of magnitude, while still achieving their same or even better query and update time efficiency. As an example, in the static setting of predecessor and range queries, the PGM-INDEX matches the query performance of a cache-optimised static B^+-TREE within *two orders of magnitude* (83×) less space; whereas in the fully-dynamic setting, where insertions and deletions are allowed, the PGM-INDEX improves the query and update time performance of a B^+-TREE by up to 71% within *three orders of magnitude* (1140×) less space.

PVLDB Reference Format:
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: https://doi.org/10.14778/3389133.3389135

This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-nd/4.0/. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 13, No. 8
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3389133.3389135

1. INTRODUCTION

The ever-growing amount of information coming from the Web, social networks and Internet of Things severely impairs the management of available data. Advances in CPUs, GPUs and memories hardly solve this problem without properly designed algorithmic solutions. Hence, much research has been devoted to dealing with this enormous amount of data, particularly focusing on algorithms for memory hierarchy utilisation [6, 35], query processing on streams [13], space efficiency [24], parallel and distributed processing [19]. But despite these formidable results, we still miss algorithms and data structures that are flexible enough to work under computational constraints that vary across users, devices and applications, and possibly evolve over time.

In this paper, we restrict our attention to the case of *indexing data structures* for internal or external memory which solve the so-called *fully-dynamic indexable dictionary* problem. This problem asks to properly store a multiset S of real keys in order to efficiently support the following query and update operations:

1. $member(x) =$ TRUE if $x \in S$, FALSE otherwise;
2. $lookup(x)$ returns the satellite data of $x \in S$ (if any), NIL otherwise;
3. $predecessor(x) = \max\{y \in S \mid y < x\}$;
4. $range(x, y) = S \cap [x, y]$;
5. $insert(x)$ adds x to S, i.e. $S \leftarrow S \cup \{x\}$;
6. $delete(x)$ removes x from S, i.e. $S \leftarrow S \setminus \{x\}$.

In the following, we use the generic expression "query operations" to refer to any of the previous kinds of pointwise queries (*member*, *lookup* and *predecessor*), while we refer explicitly to a *range* query because of its variable-size output.

A key well-known observation is that the implementation of any previous pointwise query and range operation boils down to implement the so-called $rank(x)$ primitive which returns, for any real value x, the number of keys in S which are smaller than x. In fact, if the keys in S are stored in a sorted array A, then $member(x)$ consists of just checking whether $A[rank(x)] = x$; $predecessor(x)$ consists of returning $A[rank(x) - 1]$; and $range(x, y)$ consists of scanning the array A from position $rank(x)$ up to a key greater than y.

Therefore, from now on, we focus on the implementation of *rank*, unless the formal statement of the time-space bounds requires to refer to the specific operation.

Background and related work. Existing indexing data structures can be grouped into: (i) hash-based, which range from traditional hash tables to recent techniques, like Cuckoo hashing [30]; (ii) tree-based, such as the B-TREE and its vari-

ants [3, 6, 33, 35, 37]; (iii) bitmap-based [10, 38], which allow efficient set operations; and (iv) trie-based, which are commonly used for variable-length keys. Unfortunately, hash-based indexes do not support predecessor or range searches; bitmap-based indexes can be expensive to store, maintain and decompress [36]; trie-based indexes are mostly pointer-based and, apart from recent results [15], keys are stored uncompressed thus taking space proportional to the dictionary size. As a result, the B-TREE and its variants remain the predominant data structures in commercial database systems for these kinds of queries [31].[1]

Recently, a new family of data structures, called *learned indexes*, has been introduced [22, 16]. The key idea underlying these new data structures is that indexes are *models* that we can train to learn the function *rank* that maps the keys in the input set S to their positions in the array A [1]. This parallel between indexing data structures and *rank* functions does not seem to be a new one, in fact, any of the previous four families of indexes offer a specific implementation of it. But its novelty becomes clear when we look at the keys $k \in S$ as points $(k, rank(k))$ in the Cartesian plane. As an example, let us consider the case of a dictionary of keys $a, a+1, \ldots, a+n-1$, where a is an integer. Here, $rank(k)$ can be computed *exactly* as $k - a$ (i.e. via a line of slope 1 and intercept $-a$), and thus it takes constant time and space to be implemented, independently of the number n of keys in S. This trivial example sheds light on the potential compression opportunities offered by patterns and trends in the data distribution. However, we cannot argue that all datasets follow exactly a "linear trend" and, in fact, here it comes the novel contribution of [1, 22].

These authors proposed Machine Learning (ML) techniques that learn with some "errors" the function *rank* by extracting the patterns in the data through succinct models, ranging from linear to more sophisticated ones. These errors can be efficiently corrected to return the exact value of *rank*. This way, the implementation of *rank* can be reframed as an ML problem in which we search for the model that is fast to be computed, is succinct in space, and best approximates *rank* according to some criteria that are detailed below.

This the design goal pursued by [22] with their Recursive Model Index (RMI), which uses a hierarchy of ML models organised as a Directed Acyclic Graph (DAG) and trained to learn the input distribution $(k, rank(k))$ for all $k \in S$. At query time each model, starting from the top one, takes the query key as input and picks the following model in the DAG that is "responsible" for that key. The output of RMI is the position returned by the last queried ML model, which is, however, an approximate position. A final binary search is thus executed within a range of neighbouring positions whose size depends on the prediction error of RMI.

One could presume that ML models cannot provide the guarantees ensured by traditional indexes, both because they can fail to learn the distribution and because they can be expensive to evaluate [16, 21]. Unexpectedly, RMI dominated the B$^+$-TREE, being up to 1.5–3× faster and two orders of magnitude smaller in space [22].

This notwithstanding, the RMI introduces another set of space-time trade-offs between model size and query time which are difficult to control because they depend on the distribution of the input data, on the DAG structure of RMI and on the complexity of the ML models adopted. This motivated the introduction of the FITING-TREE [17] which uses only linear models, a B$^+$-TREE to index them, and it provides an integer parameter $\varepsilon \geq 1$ to control the size of the region in which the final binary search step has to be performed. Figure 1 shows an example of a linear model f_s approximating 14 keys and its use in determining the approximate position of a key $k = 37$, which is indeed $f_s(k) \approx 7$ instead of the correct position 5, thus making an error $\varepsilon = 2$. Experimentally, the FITING-TREE improved the time performance of the B$^+$-TREE with a space saving of orders of magnitude [17], but this result was not compared against the performance of RMI. Moreover, the computation of the linear models residing in the leaves of the FITING-TREE is sub-optimal in theory and inefficient in practice. This impacts negatively on its final space occupancy and slows down its query efficiency because of an increase in the height of the B$^+$-TREE indexing those linear models.

[1]For other related work, we refer to [16, 17, 22]. Here we mention only the results that are closer to our proposal.

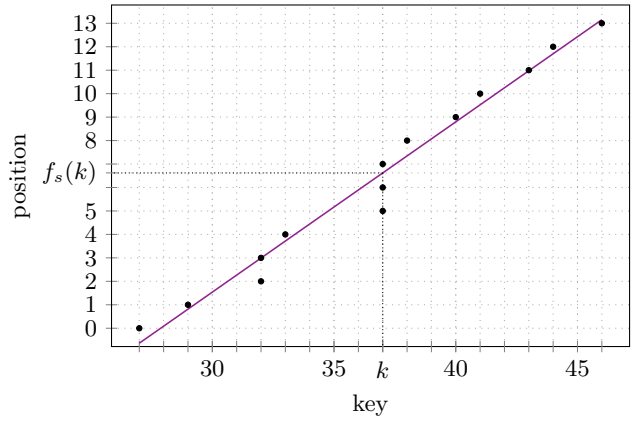


Figure 1: Linear approximation of a set of keys in the range [27, 46]. The function f_s is defined by two floats, and thus its storage is independent of the number of "covered" keys. The key $k = 37$ is repeated three times in the set, and f_s errs by $\varepsilon = 2$ in predicting the position of its first occurrence. Hence a search in a neighbourhood of size ε is enough to find the correct position of k.

Our contribution. In this paper, we contribute to the design of provably efficient learned indexes, which are also compressed, distribution-aware and auto-tuned to any given space or latency requirements.

Specifically, we design a fully-dynamic learned index, the Piecewise Geometric Model index (PGM-INDEX), which orchestrates an optimal number of linear models based on a fixed maximum error tolerance in a novel recursive structure (Sections 2 and 3). According to the lower bound proved by [32], the PGM-INDEX solves I/O-optimally the predecessor search problem while taking succinct space. Furthermore, its new design makes it a fully-learned index (unlike RMI and FITING-TREE that mix traditional and learned design elements), and it allows us to introduce novel techniques for making it compressed (Section 4) and adaptive not only to the key distribution but also to the query distribution (Section 5). We then show that the PGM-INDEX can auto-tune itself efficiently to any given space or latency requirements (Section 6).

We test the efficiency of the PGM-INDEX on synthetic and real-world datasets of up to one billion keys (Section 7). In short, the experimental achievements of the PGM-INDEX are: (i) better space occupancy than the FITING-TREE [17] by up to 75% and than the CSS-TREE [33] by a factor 83×, with the same or better query time; (ii) uniform improvement of the performance of RMI [22] in terms of query time and space occupancy, and 15× faster construction, while requiring no hyperparameter tuning; (iii) better query and update time than a B^+-TREE by up to 71% in various dynamic workloads while reducing its space occupancy by four orders of magnitude (from gigabytes to few megabytes).

As a final contribution to academic research, we release (at `https://pgm.di.unipi.it`) the implementation of our learned data structure to foster further studies and the adoption of this new generation of learned data structures in existing software.

2. PGM-INDEX

Let S be a multiset of n keys drawn from a universe $\mathcal{U}$,[2] the PGM-INDEX is a data structure parametric in an integer $\varepsilon \geq 1$ which solves the fully indexable dictionary problem on S, as defined in Section 1.

Let A be a sorted array storing the (possibly repeated) keys of S. The first ingredient of the PGM-INDEX is a Piecewise Linear Approximation model (PLA-model), namely a mapping between keys from $\mathcal{U}$ and their approximate positions in the array A. Specifically, we aim to learn a mapping that returns a position for a key $k \in \mathcal{U}$ which is at most ε away from the correct one in A. We say *piecewise* because one single linear model (also called a segment) could be insufficient to ε-approximate the positions of all the keys from $\mathcal{U}$. As a consequence, the PGM-INDEX learns a *sequence* of segments, each one taking constant space (two floats and one key) and constant query time to return the ε-approximate position of k in A. We show below in Lemma 1 that there exists a linear time and space algorithm which computes the optimal PLA-model, namely one that consists of the minimum number of ε-approximate segments. We also observe that the ε-approximate positions returned by the optimal PLA-model can be turned into *exact* positions via a binary search within a range of $\pm\varepsilon$ keys in A, thus taking time logarithmic in the parameter ε, independently of n.

The second ingredient of the PGM-INDEX is a recursive index structure that adapts to the distribution of the input keys (see Figure 2 for pictorial example). More precisely, in order to make the most of the ability of a single segment to index in constant space and time an arbitrarily long range of keys, we turn the optimal PLA-model built over the array A into a subset of keys, and we proceed recursively by building another optimal PLA-model over this subset. This process continues until one single segment is obtained, which forms the root of the PGM-INDEX.

Overall, each PLA-model forms a level of the PGM-INDEX, and each segment of that PLA-model forms a node of the data structure at that level. The speciality of this recursive construction with respect to known learned index proposals (cf. FITING-TREE or RMI) is that the PGM-INDEX is a *pure learned index* which does not hinge on classic data structures either in its structure (as in the FITING-TREE) or as a fallback when the ML models err too much (as in RMI). The net results are three main advantages that impact on its space-time complexity. First, the PGM-INDEX uses the linear models (i.e. segments) as constant-space routing tables at all levels of the data structure, while other indexes (e.g. FITING-TREE, B-TREE and variants) use space-consuming nodes storing a large number of keys which depends on the disk-page size only, thus resulting blind to the possible regularity present in the data distribution. Second, these routing tables of the PGM-INDEX take constant time to restrict the search of a key in a node to a smaller subset of keys (of size 2ε), whereas nodes in the B^+-TREE and the FITING-TREE incur a search cost that grows with the node size, thus slowing down the tree traversal during the query operations. Third, in this paper we observe that computing the minimum number of segments is a well-known computational geometry problem which admits an optimal solution in linear time and space, thus surpassing the sub-optimal proposals of FITING-TREE and RMI.

The following two subsections detail the construction and the query operations of the PGM-INDEX, while Section 3 discusses insertions and deletions.

2.1 Optimal PLA-model

In this section, we describe how an ε-approximate implementation of the mapping *rank* from keys to positions in A can be efficiently computed and succinctly stored via an optimal number of segments, which is one of the core design elements of a PGM-INDEX.

A segment s is a triple $(key, slope, intercept)$ that indexes a range of $\mathcal{U}$ through the function $f_s(k) = k \times slope + intercept$, as depicted in Figure 1. An important characteristic of the PGM-INDEX is the "precision" ε of its segments.

Definition 1. Let A be a sorted array of n keys drawn from a universe $\mathcal{U}$ and let $\varepsilon \geq 1$ be an integer. A segment $s = (key, slope, intercept)$ is said to provide an ε-*approximate* indexing of the range of all keys in $[k_i, k_{i+r}]$, for some $k_i, k_{i+r} \in A$, if $|f_s(x) - rank(x)| \leq \varepsilon$ for all $x \in \mathcal{U}$ such that $k_i \leq x \leq k_{i+r}$.

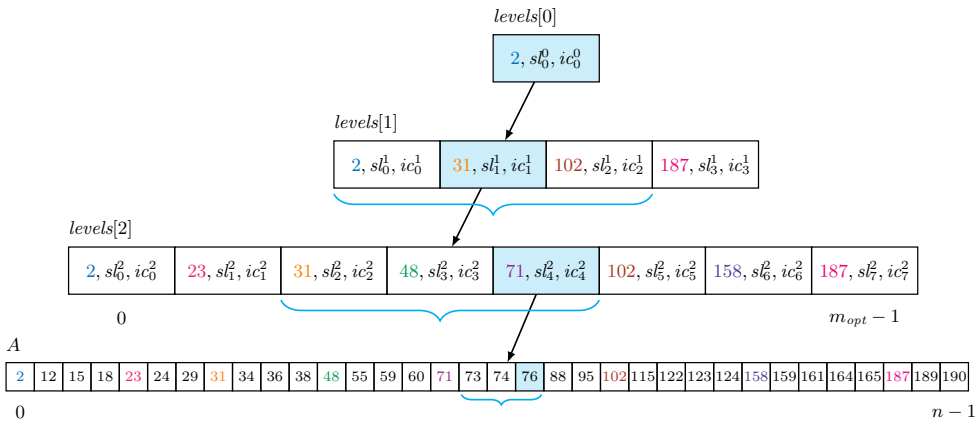
We notice that the ε-approximate indexing provided by a segment does not apply only to the keys in A but to all keys of $\mathcal{U}$. Therefore, a segment can be seen as an approximate predecessor search data structure for its covered range of keys offering $O(1)$ query time and $O(1)$ occupied space.

One segment, however, could be insufficient to ε-approximate the *rank* function over the whole set of keys in A; hence, we look at the computation of a sequence of segments, also termed PLA-model.

Definition 2. Given $\varepsilon \geq 1$, the *piecewise linear ε-approximation problem* consists of computing the PLA-model which minimises the number of its segments $\{s_0, \ldots, s_{m-1}\}$, provided that each segment s_j is ε-approximate for its covered range of keys in S. These ranges are disjoint and together cover the entire universe $\mathcal{U}$.

A way to find the optimal PLA-model for an array A is by dynamic programming, but the $O(n^3)$ time it requires is prohibitive. The authors of the FITING-TREE [17] attacked this problem via a heuristic approach, called shrinking cone,

[2] $\mathcal{U}$ is a range of reals because of the arithmetic operations required by the linear models. Our solution works for any kind of keys that can be mapped to reals while preserving their order. Examples include integers, strings, etc.




```
BUILD-PGM-INDEX(A, n, ε)
levels = an empty dynamic array
i = 0;  keys = A
repeat
    M = BUILD-PLA-MODEL(keys, ε)
    levels[i] = M;  i = i + 1
    m = SIZE(M)
    keys = [M[0].key, ..., M[m − 1].key]
until m = 1
return levels in reverse order
```

```
QUERY(A, n, ε, levels, k)
pos = f_r(k), where r = levels[0][0]
for i = 1 to SIZE(levels) − 1
    lo = max{pos − ε, 0}
    hi = min{pos + ε, SIZE(levels[i]) − 1}
    s = the rightmost segment s′ in
        levels[i][lo, hi] such that s′.key ≤ k
    t = the segment at the right of s
    pos = ⌊min{f_s(k), f_t(t.key)}⌋
lo = max{pos − ε, 0}
hi = min{pos + ε, n − 1}
return search for k in A[lo, hi]
```

Figure 2: In this PGM-index with $\varepsilon = 1$, we search for the key $k = 76$ by starting from the root segment $s' = levels[0][0] = (2, sl_0^0, ic_0^0)$ and computing the position $\lfloor f_{s'}(k) \rfloor = \lfloor k \cdot sl_0^0 + ic_0^0 \rfloor = 1$ for the next level. We then search for k in $levels[1][1-\varepsilon, 1+\varepsilon]$ among the keys $[2, 31, 102]$ (delimited by the cyan bracket), and we determine that the next segment responsible for k is $s'' = levels[1][1] = (31, sl_1^1, ic_1^1)$ because k falls between 31 and 102. Then, we compute the position $\lfloor f_{s''}(k) \rfloor = 3$, and hence we search for k in $levels[2][3-\varepsilon, 3+\varepsilon]$ among the keys $[31, 48, 71]$ (delimited by the cyan bracket). This way, we determine that $k > 71$, and hence the next segment responsible for k is $s''' = levels[2][4] = (71, sl_4^2, ic_4^2)$. Finally, we compute the position $\lfloor f_{s'''}(k) \rfloor = 17$ for the next (leaf) level (i.e. the whole array A), and hence we search for k in $A[17-\varepsilon, 17+\varepsilon]$ among the keys $[73, 74, 76]$ (delimited by the cyan bracket). Eventually, we find that k is in position 18 since $A[18] = 76$. The pseudocodes on the right formalise the construction and query algorithms of the PGM-index.

which is linear in time but does not guarantee to find the optimal PLA-model, and indeed it performs poorly in practice (as we show in Section 7.1).

Interestingly enough, we found that this problem has been extensively studied for lossy compression and similarity search of time series (see e.g. [28, 9, 11, 12, 39] and refs therein), and it admits streaming algorithms which take $O(n)$ optimal time and space. The key idea of this family of approaches is to reduce the piecewise linear ε-approximation problem to the one of constructing a convex hull of a set of points, which in our case is the set $\{(k_i, rank(k_i))\}$ grown incrementally for $i = 0, \ldots, n-1$. As long as the convex hull can be enclosed in a (possibly rotated) rectangle of height no more than 2ε, the index i is incremented and the set is extended. As soon as the rectangle enclosing the convex hull is higher than 2ε, we stop the construction and determine one segment of the PLA-model by taking the line which splits that rectangle into two equal-sized halves. Then, the current set of processed elements is emptied and the algorithm restarts from the rest of the input points. This greedy approach can be proved to be optimal in the size of the PLA-model and to have linear time and space complexity. We can rephrase this result in our context as follows.

LEMMA 1 (OPT. PLA-MODEL [28]). *Given a sequence $\{(x_i, y_i)\}_{i=0,\ldots,n-1}$ of points that are nondecreasing in their x-coordinate. There exists a streaming algorithm that in linear time and space computes the minimum number of segments that ε-approximate the y-coordinate of each point in that sequence.*

For our application to the dictionary problem, the x_is of Lemma 1 correspond to the input keys k_i, and the y_is correspond to their positions $0, \ldots, n-1$ in the sorted input array A. The next step is to prove a simple but very useful bound on the number of keys covered by a segment of the optimal PLA-model, which we deploy in the analysis of the PGM-INDEX.

LEMMA 2. *Given an ordered sequence of keys $k_i \in \mathcal{U}$ and the corresponding sequence $\{(k_i, i)\}_{i=0,\ldots,n-1}$ of points in the Cartesian plane that are nondecreasing in both their coordinates. The algorithm of Lemma 1 determines a (minimum) number m_{opt} of segments which cover at least 2ε points each, so that $m_{opt} \leq n/(2\varepsilon)$.*

PROOF. For any chunk of 2ε consecutive keys $k_i, k_{i+1}, \ldots, k_{i+2\varepsilon-1}$ take the horizontal segment $y = i + \varepsilon$. It is easy to see that those keys generate the points $(k_i, i), (k_{i+1}, i+1), \ldots, (k_{i+2\varepsilon-1}, i+2\varepsilon-1)$ and each of these keys have y-distance at most ε from that line, which is then an ε-approximate segment for that range of 2ε-keys. Hence, any segment of the optimal PLA-model covers at least 2ε keys. □

2.2 Indexing the PLA-model

The algorithm of Lemma 1 returns an optimal PLA-model for the input array A as a sequence $M = [s_0, \ldots, s_{m-1}]$ of m segments.[3] Now, in order to solve the fully indexable dictionary problem, we need a way to find the ε-approximate segment s_j responsible for estimating the approximate position pos of a query key k. This is the rightmost segment s_j such that $s_j.key \leq k$. When m is large, we could perform a binary search on the sequence M, or we could index it via a proper data structure, such as a multiway search tree (as done in the FITING-TREE). In this latter case, a query over this data structure would take $O(\log_B m + \log(\varepsilon/B))$ I/Os,

[3] To simplify the notation, we write m instead of m_{opt}.

where B is the fan-out of the multiway tree and ε is the error incurred by a segment when approximating $rank(k)$.

However, the indexing strategy above does not take full advantage of the key distribution because it resorts to a classic data structure with fixed fan-out to index M. Therefore, we introduce a novel strategy which consists of repeating the piecewise linear approximation process recursively on a set of keys derived from the sequence of segments. More precisely, we start with the sequence M constructed over the whole input array A, then we extract the first key of A covered by each segment and finally construct another optimal PLA-model over this reduced set of keys. We proceed in this recursive way until the PLA-model consists of one segment, as shown in the pseudocode of Figure 2.

If we map segments to nodes, then this approach constructs *a sort of* multiway search tree but with three main advantages with respect to B-TREEs (and thus, with respect to FITING-TREEs): (i) its nodes have variable fan-out driven by the (typically large) number of keys covered by the segments associated with those nodes; (ii) the segment in a node plays the role of a constant-space and constant-time ε-approximate routing table for the various queries to be supported; (iii) the search in each node corrects the ε-approximate position returned by that routing table via a binary search (see next), and thus it has a time cost that depends logarithmically on ε, independently of the number of keys covered by the corresponding segment.

Now, a query operation over this *Recursive* PGM-INDEX works as follows. At every level, it uses the segment referring to the visited node to estimate the position of the searched key k among the keys of the lower level.[4] The real position is then found by a binary search in a range of size 2ε centred around the estimated position. Given that every key on the next level is the first key covered by a segment on that level, we have identified the next segment to query, and the process continues until the last level is reached. The pseudocode and an example of the query operation are depicted in Figure 2.

THEOREM 1. *Let A be an ordered array of n keys from a universe $\mathcal{U}$, and $\varepsilon \geq 1$ be a fixed integer parameter. The* PGM-INDEX *with parameter ε indexes the array A in $\Theta(m)$ space and answers rank, membership and predecessor queries in $O(\log m + \log \varepsilon)$ time and $O((\log_c m)\log(\varepsilon/B))$ I/Os, where m is the minimum number of ε-approximate segments covering A, $c \geq 2\varepsilon$ denotes the variable fan-out of the data structure, and B is the block size of the external-memory model. Range queries are answered in extra (optimal) $O(K)$ time and $O(K/B)$ I/Os, where K is the number of keys satisfying the range query.*

PROOF. Each step of the recursion reduces the number of segments by a variable factor c which is at least 2ε because of Lemma 2. The number of levels is, therefore, $L = O(\log_c m)$, and the total space required by the index is $\sum_{\ell=0}^{L} m/(2\varepsilon)^{\ell} = \Theta(m)$. For the rank, membership and predecessor queries, the bounds on the running time and the I/O complexity follow easily by observing that a query performs L binary searches over intervals having size at most 2ε. In the case of range queries, we output the K keys by scanning A from the position returned by the rank query. □

[4]To correctly approximate the position of a key k falling between the last key covered by a segment s_j and the first key covered by s_{j+1}, we compute $\min\{f_{s_j}(k), f_{s_{j+1}}(s_{j+1}.key)\}$.

The main novelty of the PGM-INDEX is that its space overhead does not grow linearly with n, as in the traditional indexes mentioned in Section 1, but it depends on the "regularity trend" of the input array A. As stated in Lemma 2, the number m of segments at the last level of a PGM-INDEX is less than $n/(2\varepsilon)$. Since this fact holds also for the recursive levels, it follows that the PGM-INDEX cannot be asymptotically worse in space and time than a 2ε-way tree, such as a FITING-TREE, B$^+$-TREE or CSS-TREE (just take $c = 2\varepsilon = \Theta(B)$ in Theorem 1). Therefore, according to the lower bound proved by [32], we can state that the PGM-INDEX solves I/O-optimally the fully indexable dictionary problem with predecessor search, meaning that it can potentially replace any existing index with virtually no performance degradation.

Table 1 summarises these bounds for the PGM-INDEX and its competitors both in the Random Access Machine (RAM) and in the External Memory (EM) model for pointwise queries. The experimental results of Section 7 further support these theoretical achievements by showing that the PGM-INDEX is much faster and succinct than FITING-TREE, B$^+$-TREE and CSS-TREE because, in practice, $m_{opt} \ll n$ and the recursive structure guarantees $c \gg 2\varepsilon$.

3. DYNAMIC PGM-INDEX

Insertions and deletions in a PGM-INDEX are more difficult to be implemented compared to traditional indexes. First and foremost, the fact that a segment could index a variable and potentially large subset of data makes the classic B-TREE node split and merge algorithms inapplicable, as indeed they rely on the fact that a node contains a fixed number $\Theta(B)$ of keys. One could indeed force the segments to cover a fixed number of keys, but this disintegrates the indexing power of the PGM-INDEX. Existing learned indexes suggest inserting new elements in a sorted buffer for each node (model) which, from time to time, is merged with the main index, thus causing the retraining of the corresponding model [17, 22]. This solution is inefficient when a model indexes many keys (and thus its retraining is slow), or when the insertions hammer a certain area of the key-space (thus causing many merges due to the rapid filling of few buffers). In this section, we propose two improved strategies for handling updates, one targeted to time series, and the other targeted to more general dynamic scenarios.

If new keys are appended to the end of the array A while maintaining the sorted order (as it occurs in time series), the PGM-INDEX updates the last segment in $O(1)$ amortised time [28]. If the new key k can be covered by this last segment while preserving the ε guarantee, then the insertion process stops. Otherwise, a new segment with key k is created. The insertion of k is then repeated recursively in the last segment of the layer above. The recursion stops when a segment at any level covers k within the ε guarantee, or when the root segment is reached. At that point, the root segment might need a splitting and the creation of a new root node with its corresponding segment. Since the work at each level takes constant amortised I/Os, the overall number of I/Os required by this insertion algorithm is $O(\log_c m)$ amortised.

For inserts that occur at arbitrary positions of A, we use the logarithmic method proposed in [29, 26] and here adapted to work on learned indexes. We define a series of PGM-INDEXes built over sets $S_0, \ldots, S_b$ of keys which are

Table 1: The PGM-index improves the time, I/O and space complexity of the query operations of traditional external-memory indexes (e.g. B-tree) and learned indexes (i.e. FITing-tree). The integer parameter $\varepsilon \geq 1$ denotes the error each learned model of the PGM-index guarantees in approximating the positions of the input keys. We denote with m_{opt} the minimum number of ε-approximate segments for the input keys computed by Lemma 1 and with $m_{greedy} \geq m_{opt}$ the number of segments computed by the greedy algorithm at the core of the FITing-tree. The learned index RMI is not included in the table because it lacks guaranteed bounds.

Data structure	Space	RAM model worst case time	EM model worst case I/Os	EM model best case I/Os
Plain sorted array	$O(1)$	$O(\log n)$	$O(\log \frac{n}{B})$	$O(\log \frac{n}{B})$
Multiway tree (e.g. B-TREE)	$\Theta(n)$	$O(\log n)$	$O(\log_B n)$	$O(\log_B n)$
FITING-TREE	$\Theta(m_{greedy})$	$O(\log m_{greedy} + \log \varepsilon)$	$O(\log_B m_{greedy})$	$O(\log_B m_{greedy})$
PGM-INDEX	$\Theta(m_{opt})$	$O(\log m_{opt} + \log \varepsilon)$	$O(\log_c m_{opt})$ $c \geq 2\varepsilon = \Omega(B)$	$O(1)$

either empty or have size $2^0, 2^1, \ldots, 2^b$, where $b = \Theta(\log n)$. Each insert of a key x finds the first empty set S_i and builds a new PGM-INDEX over the merged set $S_0 \cup \cdots \cup S_{i-1} \cup \{x\}$. This union can be computed in time linear in the size of the merged set because the S_js are sorted ($0 \leq j < i$). The new sorted set consists of 2^i keys (given that $2^i = 1 + \sum_{j=0}^{i-1} 2^j$). The new merged set is used as S_i, and the previous sets are emptied. If we consider one key and examine its history over n insertions, we notice that it can participate in at most $b = \Theta(\log n)$ merges, because each merge moves the keys to the right indexes and the full $S_{\log n}$ might include all inserted keys. Given that the merges take time linear in the number of the merged keys, we pay $O(1)$ amortised time per key at each merge, that is, $O(\log n)$ amortised time per insertion overall.

The deletion of a key d is handled similarly to an insert by adding a special tombstone value that signals the logical removal of d. For details, we refer the reader to [29].

For range queries, a search in each of the S_is and a sorted merge via heaps of size b suffice, and it costs $O(\log n(\log m + \log \varepsilon) + K \log \log n)$ time, where K is the number of keys satisfying the range query. If the results of the range query can be buffered in $O(K)$ space, then the cost is reduced to $O(\log n(\log m + \log \varepsilon) + K)$ time.

In the EM model with page size B, we define instead $b' = \Theta(\log_B n)$ PGM-INDEXes built over sets of size $B^0, \ldots, B^{b'}$ and follow the ideas of [2, 35]. Summing up, we have proved the following.

THEOREM 2. *Under the same assumption of Theorem 1, the Dynamic* PGM-INDEX *with parameter ε indexes the dynamic array A and answers membership and predecessor queries in $O(\log n\,(\log m + \log \varepsilon))$ time, insertions and deletions in $O(\log n)$ amortised time. In the external-memory model with block size B, membership and predecessor queries take $O((\log_B n)(\log_c m))$ I/Os, insertions and deletions take $O(\log_B n)$ amortised I/Os, where $c \geq 2\varepsilon = \Omega(B)$ denotes the variable fan-out of the data structure. Range queries are answered with the same cost of a predecessor query plus extra $O(K)$ time and space, and $O(K/B)$ I/Os.*

We briefly mention that the bounds of Theorem 2 can be converted into worst-case bounds by keeping at most three S_i, for $1 \leq i \leq b$, and by constructing a new S_i not all at once, but spreading the construction work over the next 2^i insertions [29]. As a matter of fact, the experiments in Section 7.3 show that even the simpler amortised version is sufficient to outperform B$^+$-TREEs.

4. COMPRESSED PGM-INDEX

Compressing the PGM-INDEX boils down to providing proper lossless compressors for the keys and the segments (i.e. intercepts and slopes), which constitute the building blocks of our learned data structure. In this section, we propose techniques specifically tailored to the compression of the segments, since the compression of keys is an orthogonal problem for which there exist a plethora of solutions (see e.g. [23, 24]).

For what concerns the compression of intercepts, we proceed as follows. Intercepts can be made increasing by using the coordinate system of the segments, i.e. the one that for a segment $s_j = (key_j, slope_j, intercept_j)$ computes the position of a covered key k as $f_{s_j}(k) = (k - key_j) \times slope_j + intercept_j$. Then, since the result of $f_{s_j}(k)$ has to be truncated to return an integer position in A, we store the intercepts as integers $\lfloor intercept_j \rfloor$.[5] Finally, we exploit the fact that the intercepts are smaller than n and thus use the succinct data structure of [25] to obtain the following result.

PROPOSITION 1. *Let m be the number of segments of a* PGM-INDEX *indexing n keys drawn from a universe $\mathcal{U}$. The intercepts of those segments can be stored using $m \log(n/m) + 1.92m + o(m)$ bits and be randomly accessed in $O(1)$ time.*

The compression of slopes is more involved, and we need to design a specific novel compression technique. The starting observation is that the algorithm of Lemma 1 computes not just a single segment but a whole family of ε-approximate segments whose slopes identify an interval of reals. Specifically, let us suppose that the slope intervals for the m optimal segments are $I_0 = (a_0, b_0), \ldots, I_{m-1} = (a_{m-1}, b_{m-1})$, hence each original slope $slope_j$ belongs to I_j for $j = 0, \ldots, m-1$. The goal of our compression algorithm is to reduce the entropy of the set of these slopes by reducing their distinct number from m to t. Given the t slopes, we can store them in a table $T[0, t-1]$ and then change the encoding of each original $slope_j$ into the encoding of one of these t slopes, say $slope'_j$, which is still guaranteed to belong to I_j but now it can be encoded in $\lceil \log t \rceil$ bits (as a pointer to table T). As we will show experimentally in Section 7.4, the algorithm achieves effective compression because $t \ll m$.

Let us now describe the algorithm. First, we sort lexicographically the slope intervals I_js to obtain an array I in which overlapping intervals are consecutive. We assume

[5] Note that this transformation increases ε by 1.

that every pair keeps as satellite information the index of the corresponding interval, namely j for (a_j, b_j). Then, we scan I to determine the maximal prefix of intervals in I that intersect each other. As an example, say the sorted slope intervals are $\{(2,7),(3,6),(4,8),(7,9),\dots\}$. The first maximal sequence of intersecting intervals is $\{(2,7),(3,6),(4,8)\}$ because these intervals intersect each other, but the fourth interval $(7,9)$ does not intersect the second interval $(3,6)$ and thus is not included in the maximal sequence.

Let (l,r) be the intersection of all the intervals in the current maximal prefix of I: it is $(4,6)$ in the running example. Then, any slope in (l,r) is an ε-approximate slope for each of the intervals in that prefix of I. Therefore, we choose one real in (l,r) and assign it as the slope of each of those segments in that maximal prefix. The process then continues by determining the maximal prefix of the remaining intervals, until the overall sequence I is processed.

THEOREM 3. *Let m be the number of ε-approximate segments of a* PGM-INDEX *indexing n keys drawn from a universe $\mathcal{U}$. There exists a lossless compressor for the segments which computes the minimum number of distinct slopes $t \leq m$ while preserving the ε-guarantee. The algorithm takes $O(m \log m)$ time and compresses the slopes into $64t + m\lceil \log t \rceil$ bits of space.*

PROOF. The choice performed by the algorithm is to keep adding slope intervals in lexicographic order and updating the current intersection (l,r) until an interval (a_j, b_j) having $a_j > r$ arrives. It is easy to verify that an optimal solution has slopes within the t intersection intervals found by this algorithm. The space occupancy of the t distinct slopes in T is, assuming double-precision floats, $64t$ bits. The new slopes $slope'_j$ are still m in their overall number, but each of them can be encoded as the position $0,\dots,t-1$ into T of its corresponding double-precision float. □

5. DISTRIBUTION-AWARE PGM-INDEX

The PGM-INDEX of Theorem 1 implicitly assumes that the queries are uniformly distributed, but this seldom happens in practice. For example, queries in search engines are very well known to follow skewed distributions such as Zipf's law [38]. In such cases, it is desirable to have an index that answers the most frequent queries faster than the rare ones, so to achieve a higher query throughput [4]. Previous work exploited query distribution in the design of binary trees [7], Treaps [34], and skip lists [5], to mention a few.

In this section, we introduce a variant of the PGM-INDEX that adapts itself not only to the distribution of the input keys but also to the distribution of the queries. This turns out to be the *first distribution-aware learned index* to date, with the additional positive feature of being very succinct in space.

Formally speaking, given a sequence $S = \{(k_i, p_i)\}_{i=1,\dots,n}$, where p_i is the probability to query the key k_i (that is assumed to be known), we want to solve the distribution-aware dictionary problem, which asks for a data structure that searches for a key k_i in time $O(\log(1/p_i))$ so that the average query time coincides with the entropy of the query distribution $\mathcal{H} = \sum_{i=1,\dots,n} p_i \log(1/p_i)$.

According to [28], the algorithm of Lemma 1 can be modified so that, given a y-range for each one of n points in the plane, finds also the set of all (segment) directions that intersect those ranges in $O(n)$ time. This corresponds to finding the optimal PLA-model whose individual segments guarantee an approximation which is within the y-range given for each of those points. Therefore, our key idea is to define for every key k_i a y-range of size $y_i = \min\{1/p_i, \varepsilon\}$, and then to apply the algorithm of Lemma 1 on that set of keys and y-ranges. Clearly, for the keys whose y-range is ε we can use Theorem 1 and derive the same space bound of $O(m)$; whereas for the keys whose y-range is $1/p_i < \varepsilon$ we observe that these keys are no more than ε (in fact, the p_is sum up to 1), but they are possibly spread among all position in A, and thus they induce in the worst case 2ε extra segments. Therefore, the total space occupancy of the bottom level of the index is $\Theta(m + \varepsilon)$, where m is the one defined in Theorem 1. Now, let us assume that the search for a key k_i arrived at the last level of this Distribution-Aware PGM-INDEX, and thus we know in which segment to search for k_i: the final binary search step within the ε-approximate range returned by that segment takes $O(\log \min\{1/p_i, \varepsilon\}) = O(\log(1/p_i))$ as we aimed for.

We are left with showing how to find that segment in a distribution-aware manner, namely in $O(\log(1/p_i))$ time. We proceed similarly to the recursive construction of the PGM-INDEX, but with a careful design of the recursive step because of the probabilities (and thus the variable y-ranges) assigned to the recursively defined set of keys.

Let us consider the segment $s_{[a,b]}$ covering the sequence of keys $S_{[a,b]} = \{(k_a, p_a), \dots, (k_b, p_b)\}$, denote by $q_{a,b} = \max_{i \in [a,b]} p_i$ the maximum probability of a key in $S_{[a,b]}$, and by $P_{a,b} = \sum_{i=a}^{b} p_i$ the cumulative probability of all keys in $S_{[a,b]}$ (which is indeed the probability to end up in that segment when searching for one of its keys). To move to the next upper level of the PGM-INDEX, we create a new set of keys which includes the first key k_a covered by each segment $s_{[a,b]}$ and set its associated probability to $q_{a,b}/P_{a,b}$. Then, we construct the next upper level of the Distribution-Aware PGM-INDEX by applying the algorithm of Lemma 1 on this new set of segments. If we iterate the above analysis for this new level of "weighted" segments, we conclude that: if we know from the search executed on the levels above that $k_i \in S_{[a,b]}$, the time cost to search for k_i in this level is $O(\log \min\{P_{a,b}/q_{a,b}, \varepsilon\}) = O(\log(P_{a,b}/p_i))$.

Let us repeat this argument for another upper level to understand the influence on the search time complexity. We denote the range of keys which include k_i in this upper level with $S_{[a',b']} \supset S_{[a,b]}$, the cumulative probability with $P_{a',b'}$, and assign to the first key $k_{a'} \in S_{[a',b']}$ the probability $r/P_{a',b'}$, where r is the maximum probability of the form $P_{a,b}$ of the ranges included in $[a',b']$. In other words, if $[a',b']$ is partitioned into $\{z_1,\dots,z_c\}$, then $r = \max_{i \in [1,c)} P_{z_i, z_{i+1}}$. Reasoning as done previously, if we know from the search executed on the levels above that $k_i \in S_{[a',b']}$, the time cost to search for k_i in this level is $O(\log \min\{P_{a',b'}/r, \varepsilon\}) = O(\log(P_{a',b'}/P_{a,b}))$ because $[a,b]$ is, by definition, one of these ranges in which $[a',b']$ is partitioned.

Repeating this design until one single segment is obtained, we get a total time cost for the search in all levels of the PGM-INDEX equal to a sum of logarithms whose arguments "cancel out" (i.e. a telescoping sum) and get $O(\log(1/p_i))$.

As far as the space bound is concerned, we recall that the number of levels in the PGM-INDEX is $L = O(\log_c m)$ with $c \geq 2\varepsilon$, and that we have to account for the ε extra segments per level returned by the algorithm of Lemma 1. Consequently, this distribution-aware variant of the PGM-INDEX

takes $O(m + L\varepsilon)$ space, which is indeed $O(m)$ because ε is a constant parameter.

THEOREM 4. *Let A be an ordered array of n keys drawn from a universe $\mathcal{U}$, which are queried with (known) probability p_i, and let $\varepsilon \geq 1$ be a fixed integer parameter. The Distribution-Aware* PGM-INDEX *with parameter ε indexes the array A in $O(m)$ space and answers queries in $O(\mathcal{H})$ average time, where $\mathcal{H}$ is the entropy of the query distribution, and m is the number of segments of the optimal PLA-model for the keys in A with error ε.*

6. THE MULTICRITERIA PGM-INDEX

Tuning a data structure to match the requirements of an application is often a difficult and error-prone task for a software engineer, not to mention that these needs may change over time due to mutations in data distribution, devices, resource requirements, and so on. The typical approach is a grid search on the various instances of the data structure to be tuned until the one that matches the application's requirements is found. However, not all data structures are flexible enough to adapt at the best to these requirements, or conversely the search space can be so huge that an optimisation process takes too much time [14, 20].

In the rest of this section, we exploit the flexible design of the PGM-INDEX to show that its tuning to any space-time requirements by an underlying application can be efficiently automated via an optimisation strategy that: (i) given a space constraint outputs the PGM-INDEX that minimises its query time; or (ii) given a query-time constraint outputs the PGM-INDEX that minimises its space footprint.

The time-minimisation problem. According to Theorem 1, the query time of a PGM-INDEX can be described as $t(\varepsilon) = \delta\,(\log_{2\varepsilon} m)\log(2\varepsilon/B)$, where B is the page size of the external-memory model, m is the number of segments in the last level, and δ depends on the access latency of the memory. For the space, we introduce $s_\ell(\varepsilon)$ to denote the minimum number of segments needed to have precision ε over the keys available at level ℓ of the PGM-INDEX and compute the overall number of segments as $s(\varepsilon) = \sum_{\ell=1}^{L} s_\ell(\varepsilon)$. By Lemma 2, we know that $s_L(\varepsilon) = m \leq n/(2\varepsilon)$ for any $\varepsilon \geq 1$ and that $s_{\ell-1}(\varepsilon) \leq s_\ell(\varepsilon)/(2\varepsilon)$. As a consequence, $s(\varepsilon) \leq \sum_{\ell=0}^{L} m/(2\varepsilon)^\ell = (2\varepsilon m - 1)/(2\varepsilon - 1)$.

Given a space bound s_{max}, the "time-minimisation problem" consists of minimising $t(\varepsilon)$ subject to $s(\varepsilon) \leq s_{max}$.[6] The main challenge here is that we do not have a closed formula for $s(\varepsilon)$ but only an upper bound. Section 7.5 shows that in practice we can model $m = s_L(\varepsilon)$ with a simple power-law having the form $a\varepsilon^{-b}$, whose parameters a and b are properly estimated on the dataset at hand. The power-law covers both the pessimistic case of Lemma 2 and the best case in which the dataset is strictly linear.

Clearly, the space decreases with increasing ε, whereas the query time $t(\varepsilon)$ increases with ε since the number of keys on which a binary search is executed at each level is 2ε. Therefore, the time-minimisation problem reduces to the problem of finding the value of ε for which $s(\varepsilon) = s_{max}$ because it is the lowest ε that we can afford. Such value of ε could be found by a binary search in the bounded interval $\mathcal{E} = [B/2, n/2]$, which is derived by requiring that each model errs at least a page size (i.e. $2\varepsilon \geq B$) since lower ε values do not save I/Os, and by observing that one model is the minimum possible space (i.e. $2\varepsilon \leq n$, by Lemma 2). Furthermore, provided that our power-law approximation holds, we can speed up the search of that "optimal" ε by guessing the next value of ε rather than taking the midpoint of the current search interval. In fact, we can find the root of $s(\varepsilon) - s_{max}$, i.e. the value ε_g for which $s(\varepsilon_g) = s_{max}$. We emphasise that such ε_g may not be the solution to our problem, as it may be the case that the approximation or the fitting of $s(\varepsilon)$ by means of a power-law is not precise. Thus, more iterations of the search may be needed to find the optimal ε. Nevertheless, we guarantee to be always faster than a binary search by gradually switching to it. Precisely, we bias the guess ε_g towards the midpoint ε_m of the current search range via a simple convex combination of the two [18].

The space-minimisation problem. Given a time bound t_{max}, the "space-minimisation problem" consists of minimising $s(\varepsilon)$ subject to $t(\varepsilon) \leq t_{max}$. As for the problem above, we could binary search inside the interval $\mathcal{E} = [B/2, n/2]$ for the maximum ε that satisfies the time constraint. Additionally, we could speed up this process by guessing the next iterate of ε via the equation $t(\varepsilon) = t_{max}$, that is, by solving for ε the equation $\delta\,(\log_{2\varepsilon} s_L(\varepsilon))\log(2\varepsilon/B) = t_{max}$ in which $s_L(\varepsilon)$ is replaced by the power-law approximation $a\varepsilon^{-b}$ for proper a and b, and δ is replaced by the measured memory latency of the given machine.

Although effective, this approach raises a subtle issue, namely, the time model could not be a correct estimate of the actual query time because of hardware-dependent factors such as the presence of several caches and the CPU pre-fetching. To further complicate this issue, we note that both $s(\varepsilon)$ and $t(\varepsilon)$ depend on the power-law approximation.

For these reasons, instead of using the time model $t(\varepsilon)$ to steer the search, we measure and use the actual average query time $\overline{t}(\varepsilon)$ of the PGM-INDEX over a fixed batch of random queries. Also, instead of performing a binary search inside the whole $\mathcal{E}$, we run an exponential search starting from the solution of the dominating term $c\log(2\varepsilon/B) = t_{max}$, i.e. the cost of searching the data. Finally, since $\overline{t}(\varepsilon)$ is subject to measurement errors (e.g. due to an unpredictable CPU scheduler), we stop the search of the best ε as soon as the searched range is smaller than a given threshold.

7. EXPERIMENTS

We experimented with an implementation in C++ of the PGM-INDEX (available at github.com/gvinciguerra/PGM-index) on a machine with a 2.3 GHz Intel Xeon Gold and 192 GiB memory. We used the following three standard datasets, each having different data distributions, regularities and patterns:

- *Web logs* [22, 17] contains timestamps of about 715M requests to a web server;
- *Longitude* [27] contains longitude coordinates of about 166M points of interest from OpenStreetMap;
- *IoT* [22, 17] contains timestamps of about 26M events recorded by IoT sensors installed throughout an academic building.

[6] For simplicity, we assume that a disk page contains exactly B keys. This assumption can be relaxed by putting the proper machine- and application-dependent constants in front of $t(\varepsilon)$ and $s(\varepsilon)$.

Table 2: Space savings of PGM-index with respect to a FITing-tree for a varying ε on six synthetic datasets of 1 billion keys (improvements range from 20% to 75%) and the three real-world datasets (from 30% to 63%). We notice that these improvements are obtained without impairing the construction time because the asymptotic complexity of the two approaches is the same in theory (i.e. linear in the number of keys) and in practice (a couple of seconds, up to hundreds of millions of keys).

Dataset	ε											
	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
Uniform $u = 2^{22}$	33.8	51.0	59.4	64.0	66.5	68.1	68.6	67.4	68.8	75.4	66.7	33.3
Uniform $u = 2^{32}$	65.8	67.5	68.3	68.8	68.9	69.0	69.4	69.4	68.7	68.5	66.7	75.0
Zipf $s = 1$	47.9	54.7	59.0	61.5	62.8	60.3	44.7	33.7	29.0	28.5	27.9	28.0
Zipf $s = 2$	45.3	44.7	40.2	31.5	24.2	21.8	20.8	20.5	21.6	19.7	21.3	20.8
Lognormal $\sigma = 0.5$	66.1	67.6	68.5	68.8	68.8	68.0	62.1	46.4	35.6	32.5	30.0	29.9
Lognormal $\sigma = 1.0$	66.1	67.6	68.4	68.8	69.0	68.1	61.9	43.5	34.5	32.2	30.1	30.3
Weblogs	40.1	45.7	49.8	52.7	54.3	54.3	53.7	51.0	46.1	46.1	44.7	35.1
Longitude	59.3	63.3	62.2	56.2	49.9	45.8	43.4	40.6	38.4	38.8	38.0	39.0
IoT	46.1	48.5	50.7	47.9	46.8	41.0	37.7	44.0	43.4	34.7	30.3	58.3

We also generated some synthetic datasets according to the uniform distribution in the interval $[0, u)$, to the Zipf distribution with exponent s, and to the lognormal distribution with standard deviation σ.

7.1 Space occupancy of the PGM-index

In this set of experiments, we estimated the size of the optimal PLA-model (see Section 2.1) returned by our implementation of [39], which provides the segments stored in the bottom level of the PGM-INDEX, and compared it against the non-optimal PLA-model computed with the greedy shrinking cone algorithm used in the FITING-TREE [17]. This comparison is important because the size of a PLA-model is the main factor impacting the space footprint of a learned index based on linear models.

Table 2 shows that on synthetic datasets of 10^9 keys the improvements (i.e. relative change in the number of segments) ranged from 20% to 75%. The same table confirms these trends also for real-world datasets, on which the improvements ranged from 30% to 63%. For completeness, we report that the optimal algorithm with $\varepsilon = 8$ built a PLA-model for Web logs in 2.59 seconds, whereas it took less than 1 second for *Longitude* and *IoT* datasets. This means that the optimal algorithm of Section 2.1 can scale very fast to even larger datasets.

Since it appears difficult to prove a mathematical relationship between the number of input keys and the number of ε-approximate segments (other than the rather loose bound we proved in Lemma 2), we pursued an empirical investigation on this relation because it quantifies the space improvement of learned indexes with respect to classic indexes. Figure 3 shows that, even when ε is as little as 8, the (minimum) number m of segments is at least two orders of magnitude smaller than the original datasets size n. This reduction gets impressively evident for larger values of ε, reaching five orders of magnitude.

7.2 Query performance of the PGM-index

We evaluated the query performance of the PGM-INDEX and other indexing data structures on *Web logs* dataset, the biggest and most complex dataset available to us (see Table 2). We have dropped the comparison against the

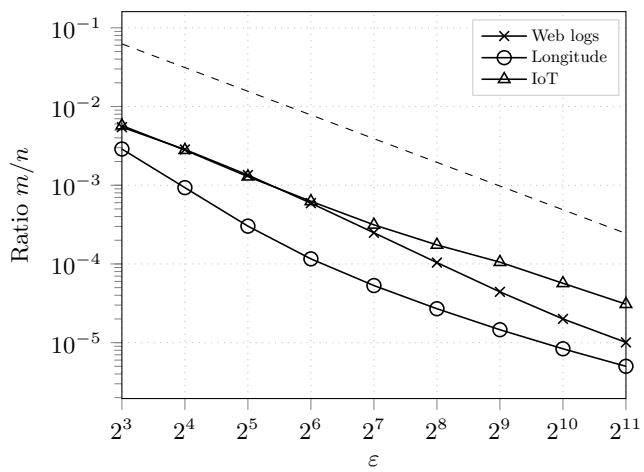


Figure 3: A log-log plot with the ratio between the number of segments m, stored in the last level of a PGM-index, and the size n of the real-world datasets as a function of ε. For comparison, the plot shows with a dashed line the function $1/(2\varepsilon)$ which is the fraction of the number of keys stored in the level above the input data of B$^+$-tree with $B = 2\varepsilon$ (see text). Note that m is 2–5 orders of magnitude less than n.

FITING-TREE, because of the evident structural superiority of the PGM-INDEX and its indexing of the optimal (minimum) number of segments in the bottom level. Nonetheless, we investigated the performance of some variants of the PGM-INDEX to provide a clear picture of the improvements determined by its recursive indexing, compared to the classic approaches based on multiway search trees (à la FITING-TREE), CSS-TREE [33] or B$^+$-TREE.

In this experiment, the dataset was loaded in memory as a contiguous array of integers represented with 8 bytes and with 128 bytes payload. Slopes and intercepts were stored as double-precision floats. Each index was presented with 10M queries randomly generated on-the-fly. The next three

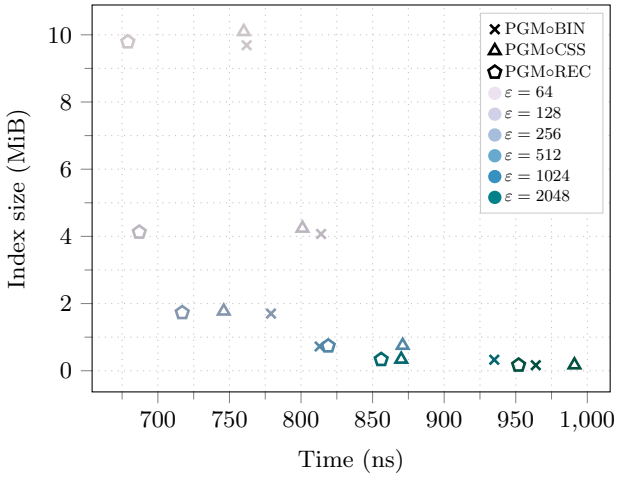


Figure 4: The (recursive) PGM-index, depicted as a pentagon, had better space and time performance than all the other configurations. For $\varepsilon > 256$ all the three configurations behaved similarly because the index was so small to fit into the L2 cache.

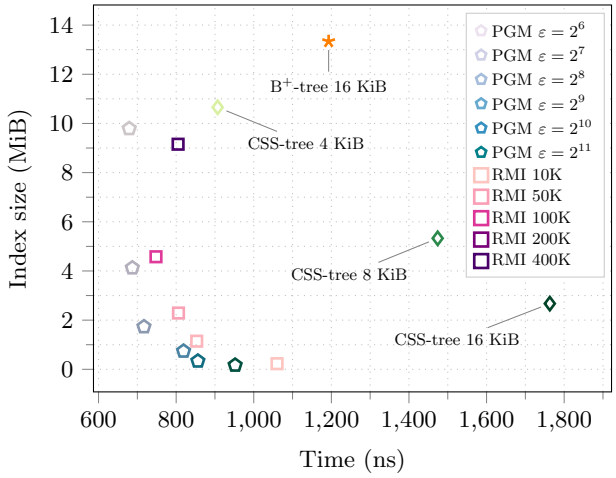


Figure 5: The PGM-index improved uniformly RMI with different second-stage sizes and traditional indexes with different page sizes over all possible space-time trade-offs. Traditional indexes with smaller page sizes are not shown because they are too far from the plot range. For example, the fastest CSS-tree occupied 341 MiB and was matched in performance by a PGM-index of only 4 MiB (83× less space); the fastest B$^+$-tree occupied 874 MiB and was matched in performance by a PGM-index which occupied only 87 KiB (four orders of magnitude less space). The construction times of CSS-tree and PGM-index were similar (1.2 and 2.1 seconds, respectively), whereas RMI took 15× more time.

paragraphs present, respectively, the query performance of the three indexing strategies for the PGM-INDEX, a comparison between the PGM-INDEX and traditional indexes, and a comparison between the PGM-INDEX and the RMI [22] under this experimental scenario.

PGM-index variants. The three indexing strategies experimented for the PGM-INDEX are binary search, multiway tree (specifically, we implemented the CSS-TREE) and our novel recursive construction (see Section 2.2). We refer to them with PGM∘BIN, PGM∘CSS and PGM∘REC, respectively. We set $\varepsilon_\ell = 4$ for all but the last level of PGM∘REC, that is the one that includes the segments built over the input dataset. Likewise, the node size of the CSS-TREE was set to $B = 2\varepsilon_\ell$ for a fair comparison with PGM∘REC. Figure 4 shows that PGM∘REC dominates PGM∘CSS for $\varepsilon \leq 256$, and has better query performance than PGM∘BIN. The advantage of PGM∘REC over PGM∘CSS is also evident in terms of index height since the former has five levels whereas the latter has seven levels, thus PGM∘REC experiences a shorter traversal time which is induced by a higher branching factor (as conjectured in Section 2.2). For $\varepsilon > 256$ all the three strategies behaved similarly because all indexes were so small that they fit into the L2 cache.

PGM-index vs traditional indexes. We compared the PGM-INDEX against the cache-efficient CSS-TREE and the B$^+$-TREE. For the former, we used our implementation. For the latter, we chose a well-known library [8, 17, 22].

The PGM-INDEX dominated these traditional indexes, as shown in Figure 5 for page sizes of 4–16 KiB. Performances for smaller page sizes were too far (i.e. worse) from the main plot range and thus are not shown. For example, the fastest CSS-TREE in our machine had page size of 128 bytes, occupied 341 MiB and was matched in query performance by a PGM∘REC with $\varepsilon = 128$ which occupied only 4 MiB (82.7× less space). As another example, the fastest B$^+$-TREE had page size of 256 bytes, occupied 874 MiB and was matched in query performance by a PGM∘REC with $\varepsilon = 4096$ and occupying 87 KiB (four orders of magnitude less space).

What is surprising in those plots is the improvement in space occupancy achieved by the PGM-INDEX which is four orders of magnitude with respect to the B$^+$-TREE and two orders of magnitude with respect to the CSS-TREE. As stated in Section 1, traditional indexes are blind to the data distribution, and they miss the compression opportunities which data trends offer. On the contrary, the PGM-INDEX is able to uncover previously unknown space-time trade-offs by adapting to the data distribution through its optimal PLA-models. For completeness, we report that on the 90.6 GiB of key-payload pairs the fastest CSS-TREE took 1.2 seconds to construct, whereas the PGM-INDEX matching its performance in 82.7× less space took only 0.9 seconds more (despite using a single-threaded and non-optimised computation of the PLA-model).

PGM-index vs known learned indexes. Figure 4 and Table 2 have shown that the PGM-INDEX improves the FITING-TREE (see also the discussion at the beginning of this section). Here, we complete the comparison against the other known learned index: namely, the 2-stage RMI which uses a combination of linear models in its two stages. Figure 5 shows that the PGM-INDEX dominates RMI, it has indeed better latency guarantees because, instead of fixing the structure beforehand and inspecting the errors afterwards, it is dynamically and optimally adapted to the input data distribution while guaranteeing the desired ε-approximation and using the least possible space. The most

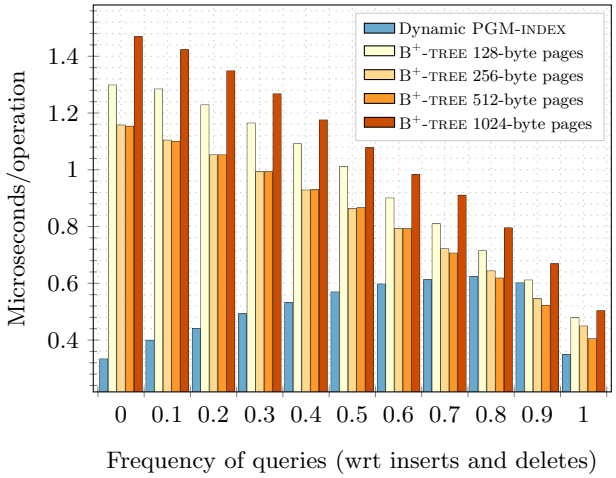


Figure 6: The Dynamic PGM-index was 13–71% faster than the B$^+$-tree in the vast majority of workloads, and it reduced the space of the four B$^+$-trees by 3891×, 2051×, 1140×, 611×, respectively.

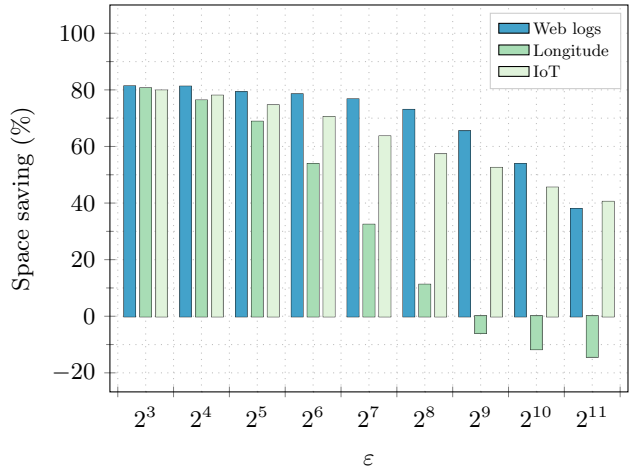


Figure 7: Slope compression reduced the space taken by the slopes by up to 81%. Longitude is the only dataset on which compression did not help for $\varepsilon \geq 2^9$ because of its special features.

compelling evidence is the Mean Absolute Error (MAE) between the approximated and the predicted position, e.g., the PGM-index with $\varepsilon = 512$ needed about 32K segments and had MAE 226±139, while an RMI with the same number of second stage models (i.e. number of models at the last level) had MAE 892±3729 (3.9×± 26.8× more). This means that RMI experienced a higher and less predictable latency in the query execution. We report that the fastest RMI took 30.4 seconds to construct, whereas the PGM-index took only 2.1 seconds (14.5× less).

Discussion. Overall, the experiments have shown that the PGM-index is fast in construction (less than 3 seconds to index a real-world table of 91 GiB with 715M key-value pairs) and has space footprint that is up to 75% lower than what is achieved by a state-of-the-art FITing-tree. Moreover, the PGM-index dominated in space and time both the traditional and other learned index structures (e.g. the RMI). In particular, it improved the space footprint of the CSS-tree by a factor 82.7× and of the B$^+$-tree by more than four orders of magnitude, while achieving the same or even better query efficiency.

7.3 Dynamic PGM-index

To experiment with insertions and deletions, we generated a dataset of 10^9 unique 8-byte keys from $u(0, 10^{12})$ and associated with each key an 8-byte value. We simulated a dynamic scenario by generating a random batch of 10M operations of which a fraction are queries, and the remaining fraction is split equally between inserts and deletes. All inserts are of new keys. For both queries and deletes, half of them refers to keys in the dataset and the other half to newly inserted items. Similarly to what we did in Section 7.2, we randomly pick the next operation from the batch instead of processing the batch sequentially. This avoids the unrealistic scenario in which all operations are known in advance and the processor speculatively executes the following operations in the batch. We fixed the ε of the PGM-index to 64 and compared it against the B$^+$-tree (since both the RMI and the CSS-tree do not support insertions and deletions).

Figure 6 shows that the Dynamic PGM-index improved by 13–71% the latency of the B$^+$-tree for the vast majority of query frequencies. The only query frequencies on which it was slower than the best-performing B$^+$-tree, i.e. the one with 512-byte pages, were 0.8 (1.0% slower) and 0.9 (15.2% slower). Overall, it is interesting to notice the latency trends of both data structures. On the one hand, in write-heavy workloads, the Dynamic PGM-index is better, whilst the B$^+$-tree pays the cost of node splits and merges. On the other hand, in read-heavy workloads, the B$^+$-tree is less penalised by those node operations and reaches similar, if not better, performance to the Dynamic PGM-index. The only exception to this latency trend occurs for a batch of only queries, on which the Dynamic PGM-index outperforms the B$^+$-tree because it is equivalent to a static PGM-index on the bulk-loaded input data.

Finally, we report that on average over the eleven query frequencies, the Dynamic PGM-index occupied 1.38 MiB, while B$^+$-trees with 128-, 256-, 512- and 1024-byte pages occupied 5.26 GiB (3890.8× more), 2.77 GiB (2050.6×), 1.54 GiB (1140.2×) and 0.83 GiB (611.1×), respectively.

7.4 Compressed PGM-index

We investigated the effectiveness of the compression techniques proposed in Section 4. Table 3 shows that the slope compression algorithm reduced the *number* of distinct slopes significantly, up to 99.94%, while still preserving the same optimal number of segments. As far as the space occupancy is concerned, and considering just the last level of a PGM-index which is the largest one, the reduction induced by the compression algorithm was up to 81.2%, as shown in Figure 7. Note that in the *Longitude* datasets for $\varepsilon \geq 2^9$ the slope compression is not effective enough. As a result, the mapping from segments to the slopes table causes an overhead that exceeds the original space occupancy of the segments. Clearly, a real-world application would turn off slope compression in such situations.

Afterwards, we measured the query performance of the Compressed PGM-index in which compression was acti-

vated over the intercepts and the slopes of the segments of all the levels. Table 4 shows that, with respect to the corresponding PGM-INDEX, the space footprint is reduced by up to 52.2% at the cost of moderately slower queries (no more than 24.5%).

Table 3: Reduction in the number of distinct slopes by the slope compression algorithm of Theorem 3.

Dataset	ε				
	8	32	128	512	2048
Weblogs	99.9	99.5	96.9	85.7	56.6
Longitude	99.2	89.0	52.6	12.8	1.3
IoT	98.4	93.2	80.7	68.0	54.4

Table 4: Query performance of the Compressed PGM-index with respect to the PGM-index.

$\varepsilon =$	64	128	256	512	1024	2048
Space saving (%)	52.2	50.8	48.5	46.0	41.5	35.5
Time loss (%)	13.7	22.6	24.5	15.1	11.7	9.9

7.5 Multicriteria PGM-index

Our implementation of the Multicriteria PGM-INDEX operates in two modes: the time-minimisation mode (shortly, min-time) and the space-minimisation mode (shortly, min-space), which implement the algorithms described in Section 6. In min-time mode, inputs to the program are s_{max} and a tolerance *tol* on the space occupancy of the solution, and the output is the value of the error ε which guarantees a space bound $s_{max} \pm tol$. In min-space mode, inputs to the program are t_{max} and a tolerance *tol* on the query time of the solution, and the output is the value of the error ε which guarantees a time bound $t_{max} \pm tol$ in the query operations. We note that the introduction of a tolerance parameter allows us to stop the search earlier as soon as any further step would not appreciably improve the solution (i.e., we seek only improvements of several bytes or nanoseconds). So *tol* is not a parameter that has to be tuned but rather a stopping criterion like the ones used in iterative methods.

To model the space occupancy of a PGM-INDEX, we studied empirically the behaviour of the number of segments $m_{opt} = s_L(\varepsilon)$ forming the optimal PLA-model, by varying ε and by fitting ninety different functions over about two hundred points $(\varepsilon, s_L(\varepsilon))$ computed beforehand on our real-world datasets. Looking at the fittings, we chose to model $s_L(\varepsilon)$ with a power-law having the form $a\varepsilon^{-b}$. As further design choices we point out that: (i) the fitting of the power-law was performed with the Levenberg-Marquardt algorithm, while root-finding was performed with Newton's method; (ii) the search space for ε was set to $\mathcal{E} = [8, n/2]$ (since a cache line holds eight 64-bit integers); and finally (iii) the number of guesses was set to $2\lceil \log\log \mathcal{E} \rceil$.

The following experiments were executed by addressing some use cases in order to show the efficacy and efficiency of the Multicriteria PGM-INDEX.

Experiments with the min-time mode. Suppose that a database administrator wants the most efficient PGM-INDEX for the *Web logs* dataset that fits into an L2 cache of 1 MiB. Our solver derived a PGM-INDEX with minimum query latency and that space bound by setting $\varepsilon = 393$ and taking 10 iterations in 19 seconds. This result was obtained by approximating $s_L(\varepsilon)$ with the power-law $46032135 \cdot \varepsilon^{-1.16}$ which guaranteed a mean squared error of no more than 4.8% over the range $\varepsilon \in [8, 1024]$.

As another example, suppose that a database administrator wants the most efficient PGM-INDEX for the *Longitude* dataset that fits into an L1 cache of 32 KiB. Our solver derived a PGM-INDEX with minimum query latency and that space bound by setting $\varepsilon = 1050$ and taking 14 iterations in 9 seconds.

Experiments with the min-space mode. Suppose that a database administrator wants the PGM-INDEX for the IoT dataset with the lowest space footprint that answers any query in less than 500 ns. Our solver derived an optimal PGM-INDEX matching that time bound by setting $\varepsilon = 432$, occupying 74.55 KiB of space, and taking 9 iterations and a total of 6 seconds.

As another example, suppose that a database administrator wants the most compressed PGM-INDEX for the *Web logs* dataset that answers any query in less than 800 ns. Our solver derived an optimal PGM-INDEX matching that time bound by setting $\varepsilon = 1217$, occupying 280.05 KiB of space, and taking 8 iterations and a total of 17 seconds.

Discussion. In contrast to the FITING-TREE and the RMI, the Multicriteria PGM-INDEX can trade efficiently query time with space occupancy, making it a promising approach for applications with rapidly-changing data distributions and space/time constraints. Overall, in both modes, our approach ran in less than 20 seconds.

8. CONCLUSIONS AND FUTURE WORK

We have introduced the PGM-INDEX, a learned data structure for the fully-dynamic indexable dictionary problem that improves the query/update performance and the space occupancy of both traditional and modern learned indexes up to several orders of magnitude. We have also designed three variants of the PGM-INDEX: one that improves its already succinct space footprint using ad-hoc compression techniques, one that adapts itself to the query distribution, and one that efficiently optimises itself within some user-given constraint on the space occupancy or the query time.

We leave as future work the implementation and experimentation of the Distribution-Aware PGM-INDEX and the study of a mapping between the input keys and the space occupancy of the resulting index (which besides the theoretical importance would boost the performance of our Multicriteria PGM-INDEX). Additional opportunities include the integration and the experimentation of the PGM-INDEX into a real DBMS, especially in the external-memory scenario, and the study of alternative implementations (e.g. with the use of SIMD instructions in the search/update algorithms).

Acknowledgements. Part of this work has been supported by the Italian MIUR PRIN project "Multicriteria data structures and algorithms: from compressed to learned indexes, and beyond" (Prot. 2017WR7SHH), by Regione Toscana (under POR FSE 2014/2020), by the European Integrated Infrastructure for Social Mining and Big Data Analytics (SoBigData++, Grant Agreement #871042), and by PRA UniPI 2018 "Emerging Trends in Data Science".

9. REFERENCES

[1] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[2] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2):147 – 162, 2004.

[3] M. Athanassoulis and A. Ailamaki. BF-tree: approximate tree indexing. *PVLDB*, 7(14):1881–1892, 2014.

[4] M. Athanassoulis, K. S. Bøgh, and S. Idreos. Optimal column layout for hybrid workloads. *PVLDB*, 12(13):2393–2407, 2019.

[5] A. Bagchi, A. L. Buchsbaum, and M. T. Goodrich. Biased skip lists. *Algorithmica*, 42(1):31–48, 2005.

[6] M. Bender, E. Demaine, and M. Farach-Colton. Cache-oblivious B-trees. *SIAM Journal on Computing*, 35(2):341–358, 2005.

[7] S. W. Bent, D. D. Sleator, and R. E. Tarjan. Biased search trees. *SIAM Journal on Computing*, 14(3):545–568, 1985.

[8] T. Bingmann. STX B^+-tree C++ template classes, 2013. `http://panthema.net/2007/stx-btree`, v0.9.

[9] C. Buragohain, N. Shrivastava, and S. Suri. Space efficient streaming algorithms for the maximum error histogram. In *Proceedings of the IEEE 23rd International Conference on Data Engineering*, ICDE, pages 1026–1035, Washington, D.C., USA, 2007.

[10] C.-Y. Chan and Y. E. Ioannidis. Bitmap index design and evaluation. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, pages 355–366, New York, NY, USA, 1998.

[11] D. Z. Chen and H. Wang. Approximating points by a piecewise linear function. *Algorithmica*, 66(3):682–713, 2013.

[12] Q. Chen, L. Chen, X. Lian, Y. Liu, and J. X. Yu. Indexable PLA for efficient similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB, pages 435–446, 2007.

[13] G. Cormode. Data sketching. *Communications of the ACM*, 60(9):48–55, 2017.

[14] N. Dayan, M. Athanassoulis, and S. Idreos. Optimal bloom filters and adaptive merging for LSM-trees. *ACM Transactions on Database Systems*, 43(4):16:1–16:48, Dec. 2018.

[15] P. Ferragina and R. Venturini. Compressed cache-oblivious String B-tree. *ACM Transactions on Algorithms*, 12(4):52:1–52:17, 2016.

[16] P. Ferragina and G. Vinciguerra. *Learned data structures*. In "Recent Trends in Learning From Data", Eds. L. Oneto et al., Studies in Computational Intelligence, pages 5–41. Springer, 2020.

[17] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska. FITing-tree: A data-aware index structure. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, pages 1189–1206, New York, NY, USA, 2019.

[18] G. Graefe. B-tree indexes, interpolation search, and skew. In *Proceedings of the 2nd International Workshop on Data Management on New Hardware*, DaMoN, New York, NY, USA, 2006.

[19] A. Grama, G. Karypis, V. Kumar, and A. Gupta. *Introduction to Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2003.

[20] S. Idreos, K. Zoumpatianos, S. Chatterjee, W. Qin, A. Wasay, B. Hentschel, M. Kester, N. Dayan, D. Guo, M. Kang, and Y. Sun. Learning data structure alchemy. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 42(2):46–57, 2019.

[21] T. Kraska, M. Alizadeh, A. Beutel, E. H. Chi, A. Kristo, G. Leclerc, S. Madden, H. Mao, and V. Nathan. SageDB: A learned database system. In *Proceedings of the 9th Biennial Conference on Innovative Data Systems Research*, CIDR, 2019.

[22] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, pages 489–504, New York, NY, USA, 2018.

[23] A. Moffat and A. Turpin. *Compression and Coding Algorithms*. Springer, Boston, MA, USA, 2002.

[24] G. Navarro. *Compact data structures: A practical approach*. Cambridge University Press, New York, NY, USA, 2016.

[25] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proceedings of the SIAM Meeting on Algorithm Engineering & Expermiments*, pages 60–70, Philadelphia, PA, USA, 2007.

[26] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, June 1996.

[27] OpenStreetMap contributors. OpenStreetMap Data Extract for Italy. `https://www.openstreetmap.org`, 2018. Retrieved from `http://download.geofabrik.de` on May 9, 2018.

[28] J. O'Rourke. An on-line algorithm for fitting straight lines between data ranges. *Communications of the ACM*, 24(9):574–578, 1981.

[29] M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes in Computer Science*. Springer, 1983.

[30] R. Pagh and F. F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122 – 144, 2004.

[31] A. Petrov. Algorithms behind modern storage systems. *Communications of the ACM*, 61(8):38–44, 2018.

[32] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th ACM Symposium on Theory of Computing*, STOC, pages 232–240, New York, NY, USA, 2006.

[33] J. Rao and K. A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB, pages 78–89, San Francisco, CA, USA, 1999.

[34] R. Seidel and C. R. Aragon. Randomized search trees. *Algorithmica*, 16(4/5):464–497, 1996.

[35] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Computing Surveys*, 33(2):209–271, 2001.

[36] J. Wang, C. Lin, Y. Papakonstantinou, and S. Swanson. An experimental study of bitmap compression vs. inverted list compression. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, pages 993–1008, New York, NY, USA, 2017.

[37] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a Bw-tree takes more than just buzz words. In *Proceedings of the ACM International Conference on Management of Data*, SIGMOD, pages 473–488, New York, NY, USA, 2018.

[38] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 1999.

[39] Q. Xie, C. Pang, X. Zhou, X. Zhang, and K. Deng. Maximum error-bounded piecewise linear representation for online stream approximation. *The VLDB Journal*, 23(6):915–937, 2014.