Learned data structures

Paolo Ferragina and Giorgio Vinciguerra

Abstract Very recently, the unexpected combination of data structures and machine learning has led to the development of a new area of research, called *learned data structures*. Their distinguishing trait is the ability to reveal and exploit patterns and trends in the input data for achieving more efficiency in time and space, compared to previously known data structures. The goal of this chapter is to provide the first comprehensive survey of these results and to stimulate further research in this promising area.

1 Introduction

The *searching problem* is among the oldest and most prominent problems in computer science, well-studied and ubiquitous in research and applications. Not surprisingly, it is often used as an introductory topic in basic algorithms courses, paving the way to the study of fundamental data structures such as arrays, lists, search trees, tries and hash tables.

Simply stated, the searching problem asks to preprocess a set of items into a data structure in such a way that certain kinds of queries on these items can subsequently be answered quickly. Clearly, there are many different facets and solutions to this problem, depending on how the data are organised in memory and which operations are supported, but the above basic data structures are often sufficient to solve sophisticated versions of the problem, be them a search of neighbouring points of interest in a map or a search for all documents relevant to a query in a search engine. As an example, a data structure for this last application, called inverted list, consists

Paolo Ferragina
Università di Pisa, e-mail: `paolo.ferragina@unipi.it`

Giorgio Vinciguerra
Università di Pisa, e-mail: `giorgio.vinciguerra@phd.unipi.it`

of a *hash table* or a *trie* mapping each possible query term to a *list* or an *array* of document-IDs containing that term.

We refer to this first family as *traditional data structures*, which also includes variations (stacks, queues, heaps, circular lists, . . .) and combinations (multiway trees, spatial data structures, randomised lists, . . .) of the most known ones. In these data structures, the main goal pursued by researchers and software engineers was to achieve efficient/optimal query time and space as a function of the *number* of items [6].

Since the '90s, with the flood of big data and the advent of hierarchical memories in computers, researchers aimed at designing *compact data structures*, sometimes called succinct, compressed or opportunistic data structures. These data structures exploit the repetitiveness present in the input data to occupy a space close to the information-theoretic lower bound and still provide efficient query operations. This means that they do not require that data is fully decompressed to perform searches over them. Today, it is known how to turn almost any traditional data structure into a compact data structure [27].

In general, both families of traditional and compact data structures offer a wide range of trade-offs, and no single solution is satisfactory for every application because of differing hardware and software requirements or constraints imposed by user needs. As a result, software engineers have often to choose (sometimes incautiously) one among a multitude of data structures based on partial or superficial information available when their choice is done, and then soon discover that their choice is inefficient because it did not take into account some specialities of input data. This situation is well captured by the following excerpt:

> Another simple way to facilitate [. . .] retrieval is to let *people* do part of the work, by providing them with suitable printed indexes to the information. This method is often the most reasonable and economical way to proceed (provided, of course, that the old paper is recycled whenever a new index is printed), especially because people tend to notice interesting patterns when they have convenient access to masses of data.
>
> — Donald Knuth, *The Art of Computer Programming* (1973).

Recently, researchers were able to define machine learning-based tools that automatically detect such patterns and, unexpectedly, orchestrated them with classic data structures to design a new family of data structures, called *learned data structures*, that attempts exactly to reveal and exploit patterns and trends in the input data for achieving more efficiency in time and space compared to the previously known solutions. The key design idea consists of augmenting —and sometimes even replacing — classic data-structural building blocks, such as tree nodes or hash tables, with Machine Learning (ML) models which are better suitable to *"notice interesting patterns when they have convenient access to masses of data"*. This feature, combined with proper data structural design elements and algorithms has led to outstanding improvements in space occupancy and time efficiency over a plethora of searching problems, some of which will be introduced and discussed in the following sections.

A first overview of the problems and corresponding results achieved in this new algorithmic field is offered in Table 1, where for each problem we summarise the

Table 1 A summary of machine-learned data structures for four main searching problems. CDF stands for Cumulative Density Function of a probability distribution.

	Range queries Section 2	Exact membership Section 3	Approx. membership Section 4	Frequency estimation Section 5
Approach	Learn CDF of data and use it to map queries to memory locations	Learn CDF of data and use it as a hash function	Classify the membership of items	Classify heavy hitters and/or predict the frequency of items
Pros	Reduced space and faster queries	Less hash collisions	Reduced space	Improved estimations
Cons	No clear extensions to variable-length keys	Increased space for the hash function, and increased query latency. Susceptible to adversarial attacks	No guarantees if distribution changes. Modest space reductions. Susceptible to adversarial attacks	Requires historical data
Refs	[10, 11, 13, 14, 21, 22, 32, 34, 30, 35, 23]	[21, 36]	[9, 21, 24, 26, 31]	[16, 37]

ML approach taken, its benefits and drawbacks, and we point out the main references to the literature. We strictly limit ourselves to four main searching problems: exact membership, range queries, approximate membership, and frequency estimation. The set of problems which can be solved via ML-based approaches is growing — just to mention a few, cardinality estimation of SQL queries via deep learning [19], learned query optimisers [20], learned operating systems [38], learned sorting algorithms [20], learned prefetchers [15] — but they are beyond the scope of this chapter. Furthermore, while not strictly related to the content of this chapter, we ought to mention the work of [18, 17] concerning the semi-automated (and possibly ML-driven) design of data structures from their first design principles.

Notation and basic terminology. We denote by n the input size and use log to denote the logarithm to the base 2. To analyse algorithms, we use both the Random Access Machine (RAM) model and the external (or, two-level) memory model [33]. The RAM model consists of an infinite memory of $O(\log n)$ bit cells, and it supports arithmetic, logical and bitwise operations on individual cells in constant time. The time cost of an algorithm is evaluated by counting the asymptotic number of steps it takes to solve a problem, while its space cost is the maximal number of memory cells (sometimes expressed in bits) it occupies during the computation. On the other hand, the external memory model abstracts the memory hierarchy by modelling just two levels: an internal memory of limited size M, and an external memory of unlimited size divided into blocks of B consecutive items. Data is brought into internal memory and written back to external memory by transferring one block at a time. The efficiency of an algorithm is then evaluated by counting the asymptotic number of transfers, or I/Os, it makes for solving a given problem.

2 Learned data structures for range queries

Structuring data to provide fast retrieval by individual keys or range of keys is a problem as old as computer science, arguably the most successful example of the interplay between data structures and machine learning.

The *indexable dictionary problem* asks to store a multiset S of keys drawn from a universe $\mathcal{U}$ in order to efficiently support the following query and update operations:

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

A data structure implementing the above query and update operations is called an *index structure*, or simply *index*. The B-tree and its variations are from the '70s the predominant indexes for working in disk memories in commercial database systems [29]. A B-tree is a search tree with fan-out $\Theta(B)$. A node r stores $r.num = \Theta(B)$ keys $r.key_i$ in ascending order and associated data $r.data_i$ ($1 \leq i \leq r.num$). An internal node r stores also $r.num + 1$ pointers to its children $r.child_j$ ($1 \leq j \leq r.num + 1$). A query *lookup*(x) starts from the root being the current node. If this node contains a key equal to x, then the search terminates successfully and returns the data associated with x. Otherwise, if the node is a leaf, the search terminates unsuccessfully and returns NIL. In the other cases, the search recurses into the unique children that may contain x. Since a node can be accessed in $O(1)$ I/Os, the cost of a search is proportional to the height of the tree, i.e. $O(\log_B n)$ I/Os. Other pointwise queries are slight modifications of the tree traversal we just described.

A variation of the B-tree, called B$^+$-tree, stores instead the keys from S and any associated data at the leaves, which are linked left-to-right for fast range queries. Inner nodes contain copies of certain keys from S and act only as routing elements, i.e. an inner node r contains keys $r.key_i$ and pointers to children $r.child_j$ ($1 \leq i < j \leq r.num + 1$) but not the data associated with keys, as shown in Figure 1. A *lookup* in a B$^+$-tree costs $\Theta(\log_B n)$ I/Os.

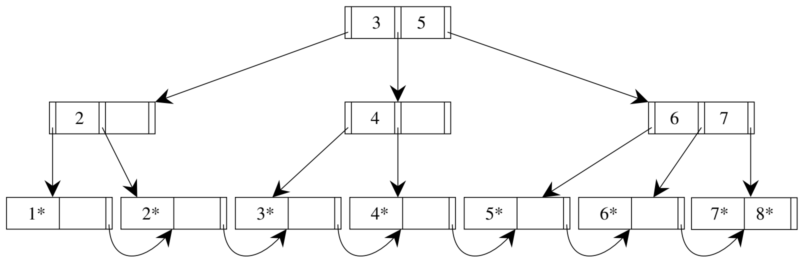

Fig. 1 A B$^+$-tree with fan-out 3 stores all the keys $x = 1, 2, \ldots, 8$ and associated data, denoted as x^*, at the leaf level. Inner nodes store copies of certain keys and act as routing elements.

The work of Kraska et al. [21], which extended some previous original results of Ao et al. [1], has provided us with a different perspective on this old-fashioned problem. The key idea introduced by these authors is that *indexes are models* that can be *trained* to map keys to their location in the sorted order, and this mapping is enough to efficiently implement any pointwise and range query of the indexable dictionary problem. In fact, let us denote by $rank(x)$ the primitive that returns, for any key $x \in \mathcal{U}$, the number of keys in S which are smaller than x, and let A be the array storing the keys of S in sorted order. Then, $member(x)$ can be implemented by checking whether $A[rank(x)] = x$ or not; $predecessor(x)$ consists of returning $A[rank(x)-1]$; and $range(x, y)$ consists of scanning the array A from position $rank(x)$ up to the first key larger than y.

This parallel between index structures and *rank* function does not seem a new one, as indeed any B-tree offers an implementation of it. But its novelty becomes clear when we look at the keys $x \in S$ as points $(x, rank(x))$ in the Cartesian plane. As an example, let us consider the case of a set of keys $S = \{2, 4, 6, \ldots, 2n\}$. Here, as depicted in Figure 2, the points $(x, rank(x))$ can be "covered" by the linear model $f(x) = 0.5x - 1$ so that the function $rank(x)$ can be computed exactly for each key $x \in S$ in constant time and space, independently of the number of keys in S. Consequently, one should never build a B-tree on that set of keys!

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, nor that the models can learn the data distribution with no errors. Nevertheless, the idea is promising so that, to deal with the general setting, we have to design techniques that:

- learn the *rank* function by extracting the patterns in the data through succinct models, ranging from linear to more sophisticated ones;
- admit some "errors" in the output of the model approximating the *rank* function and which, in turn, can be efficiently (in time and space) corrected to return the exact value of *rank*.

This is a supervised learning task in which the dataset $D = \{(x, rank(x))\}_{x \in S}$ is given, and we look for a model $f : \mathcal{U} \to [0, n)$ which maps keys of S to their positions in the

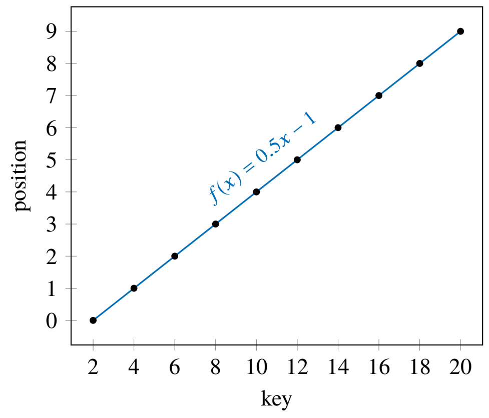

Fig. 2 A set of keys $S = \{2, 4, 6, \ldots, 2n\}$ mapped to points $(x, \mathit{rank}(x))$ in the Cartesian plane, and the line passing through ("covering") all of them.

sorted order, and minimises the error $|f(x) - \mathit{rank}(x)|$ over all $x \in S$.[1] Equivalently, we look for a model $F\colon \mathcal{U} \to [0, 1]$ which minimises $|nF(x) - \mathit{rank}(x)|$ over all $x \in S$. Intuitively, $F(x)$ is the fraction of keys that are less than x, i.e. the learned empirical Cumulative Distribution Function (CDF) of D. It is clear that $f(x)$ or $F(x)$ alone are not sufficient to solve the searching problem for a key x in S because of the errors present in the learned approximation provided by those functions. Hence, proper algorithms and data structures have to be designed to orchestrate learned models with classic building blocks that allow to *correct* these approximations and provide correct answers to the searching for x.

A simple but illustrative example of such algorithmic corrections follows below.

Example: implementing *rank* with a linear regression model

Let us given an array $A = [27, 29, 32, 34, 34, 35, 37, 37, 37, 38, 38, 40, 41]$ of 13 integer (repeated) keys, and consider the corresponding set of points in the Cartesian plane obtained by representing only the first occurrence of every key with its *rank* (but dropping the other copies):

$$\begin{aligned} D &= \{(x, \mathit{rank}(x))\}_{x \in A} \\ &= \{(27, 0), (29, 1), (32, 2), (34, 3), (35, 5), (37, 6), (38, 9), (40, 11), (41, 12)\}. \end{aligned}$$

The linear model f computed using ordinary least squares on D has slope 0.88 and intercept -25.23. As shown in Figure 3, if we use f to approximate the rank of the key $x = 34$, we get $r = \lceil f(x) \rceil = \lceil 4.69 \rceil = 5$, but the true rank of x is 3. We can fix the error incurred by f via a linear search that starts from $A[r]$ and stops when the first occurrence of (a key less than or equal to) x is found.

[1] We assume that universe $\mathcal{U}$ is a range of reals because of the arithmetic operations required by the models. This works for any kind of keys that can be mapped to reals by preserving their order, such as integers or strings.

Furthermore, instead of using a linear search, we can keep the maximum error *err* incurred by f over the key in S, and use it at query time to perform a binary search in $A[r - err, r + err]$ in $O(\log err)$ time. Likewise, an *exponential search* (also called *doubling search* or *galloping search* [4]) starting from $A[r]$ could solve the problem more efficiently in time $O(\log d)$, where $d = |rank(x) - r|$ is the actual distance between the estimated position and the correct position of x in A.

Finally, we notice that this approach based on linear models and algorithmic correction can also be used to support unsuccessful searches for keys not in A with the same time complexity. As an example, let us assume that we wish to search for $x = 33$. We compute $\lceil f(x) \rceil = \lceil 0.88 \times 33 - 25.23 \rceil = \lceil 3.81 \rceil = 4$ and then start an exponential search towards the beginning of A (since $A[4] = 34 > 33$), which finds that 33 does not occur in A.

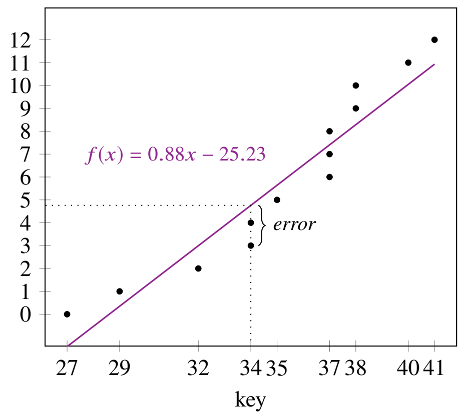

Fig. 3 Real data rarely have trivial trends like the one in Figure 2. More often there are missing keys (gaps in the horizontal axis) and repeated keys (points stacked vertically). In such cases, before replacing a learned model with a traditional index we must design a strategy to fix the error of the model.

In the following sections, we trace the logical development of more sophisticated learned data structures. We look at different techniques to learn the function *rank* for the keys in S, and present solutions to the challenges that arise when replacing well-established algorithms and data structures with ML models in several searching problems, starting from the classic range search.

2.1 The Recursive Model Index

The Recursive Model Index (RMI) [21] is a fixed hierarchy of regression models organised in stages. At query time each model, starting from the one at the root, takes the searched key as input and picks the model in the lower stage responsible for that key, i.e. the expert of a certain range of the training data where the key falls

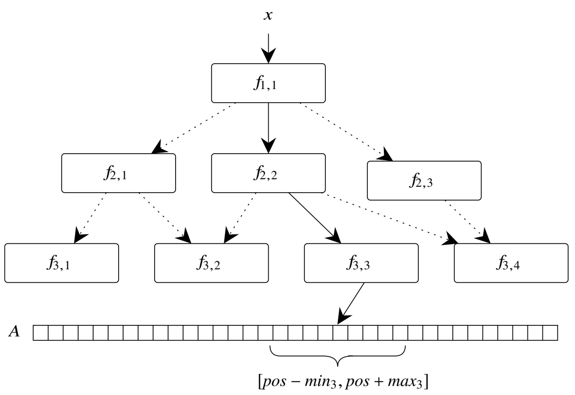

Fig. 4 RMI is a fixed hierarchy of models organised in stages. In this example, the root model $f_{1,1}$ routes the input key x to the model $f_{2,2}$ in the second stage, which in turn routes x to the last-stage model $f_{3,3}$. The output of $f_{3,3}$ along with its errors min_j and max_j is used to limit the final search step to $A[pos - min_j, pos + max_j]$.

into. The output of a model in the last stage is used as an approximate position of the searched key in A, the sorted array of input keys.

If we imagine creating an edge from each model to the models it picks in the stage below, then the resulting structure is a Direct Acyclic Graph (DAG) since different models at one stage can pick the same model at the stage below, as pictured in Figure 4.

The construction of RMI proceeds top-down by training the hierarchy of regression models. First, the root model $f_{1,1}$ is trained on the entire dataset D. Second, this model is used to distribute each pair $(x, y) \in D$ to one of the p models in the lower stage according to the map $\lfloor p \cdot f_{1,1}(x)/n \rfloor$, thus effectively partitioning D into $D_1, \dots, D_p$. Third, the process is repeated recursively by training the jth model $f_{2,j}$ on the dataset D_j for all $j = 1, \dots, p$. This way, keys are further partitioned in the models on the lower stage, and the process repeats recursively for all ℓ stages (i.e. the depth of the DAG). The time to build RMI, by assuming that the training time of a model is linear in the size of the training set, is $O(\ell n)$.

After the training, the worst over-prediction and under-prediction of every model $f_{\ell,j}$ in the last stage are evaluated (and stored) as

$$min_j = \Big| \min_{(x,y)\in D_{\ell,j}} (y - f_{\ell,j}(x)) \Big| \quad \text{and} \quad max_j = \Big| \max_{(x,y)\in D_{\ell,j}} (y - f_{\ell,j}(x)) \Big|. \tag{1}$$

If one of the two errors exceeds a certain user-defined threshold, the corresponding model is replaced by a B-tree. Otherwise, the errors are stored and used at query time to limit the final binary-search step to $A[pos - min_j, pos + max_j]$, where pos is the approximate position computed via the jth last-stage model.

An example of searching for a key x in RMI is shown in Figure 4. Note that RMI has to be monotonic to ensure that the errors min_j and max_j are guaranteed also for the keys not in D.

The recursive model index is different from traditional tree-based indexes (such as B-trees) for at least four main reasons:

1. Its size is constant and is given by the overall number of parameters in the hierarchy of models plus the two errors stored for each model in the last stage. Conversely, the size of traditional indexes is linear in n.
2. It can take advantage of patterns and trends in the distribution of the input data. Conversely, traditional indexes are general-purpose and do not make assumptions about the data distribution.
3. In-between stages there are no searches, as the output of a model is directly used to pick the model of the next stage. Conversely, traditional indexes perform a search to identify the next node to visit.
4. The error cannot be bounded/evaluated beforehand. Conversely, traditional indexes allow specifying the node size (i.e. the maximum number of keys inspected at each level) thus bounding in advance the worst-case performance.

On datasets of 200 million entries, a two-stage RMI[2] always dominated an in-memory implementation of a B^+-tree, being up to 1.5–3× faster and up to two orders of magnitude smaller.[3]

Among the shortcomings of RMI, we mention: (i) the lack of support for insertions and deletions; (ii) the time-consuming tuning process, needed to shape the hierarchy of models, and the subsequent training/construction time; (iii) the lack of latency guarantees; (iv) the top-down training algorithm, which blindly distributes keys to the models below ignoring their power or workload in terms of partition size.

2.2 Variations and extensions of RMI

After the introduction of RMI, subsequent research has focused mainly on its fixed structure and the lack of support for insertions and deletions of keys. A common denominator of the subsequent research efforts is “adaptability”, that is, getting rid of the fixed hierarchical structure of RMI, which without a proper (time-consuming) tuning can lead to redundant or overloaded models. Redundancy occurs when upper stages of RMI distribute too few keys to a single model, thus failing to use its approximation potential at full scale. Overloading occurs when upper stages distribute too many keys to a single model, which may not have the capability of fully capture the data trend, thus causing high prediction errors and in turn high query times.

In the following, we present four learned indexes that address the problems mentioned above.

[2] Private correspondence with the authors clarified that only linear models were used in the hierarchy due to their superior performance.

[3] It has to be noted that a B^+-tree supports also insertions and deletions, which require a more complex structure that is both slower and more space-inefficient than other static tree-based indexes.

2.2.1 ASLM

The Adaptive Single Layer Model (ASLM) [23] is a single-stage learned index that addresses two shortcomings of RMI: the potentially poor distribution of keys among models in the lower levels of the DAG, and the lack of support for updates.

For the first problem, ASLM heuristically partitions the dataset D to minimise the chance that a model is trained on data points which are far apart. First, the Euclidean distance between every two consecutive points in the sorted dataset D is computed. Second, the distances are sorted in descending order. Third, the points corresponding to the first p distances in the sorted order are selected as partition boundaries, where p is a fixed parameter. Fourth, the partitions are further refined by splitting or merging the consecutive ones whose size is above or below a certain fixed threshold, so to guarantee a certain balance among the sizes of the partitions. Finally, a model is created for each partition and trained on it. Overall, the construction cost is bounded above by the sorting of n keys.

ASLM requires some sort of search to select the correct model for the queried key, while RMI uses the output of the root model to pick the model in the next stage. Moreover, the adaptability of ASLM to a given dataset is somehow hindered by the fact that the number p of models and the thresholds for splitting/merging partitions must be tuned. Authors reported that the partitioning strategy of ASLM reduced the average prediction error up to 3.8× with respect to a two-stage RMI that used 3-layer neural networks with 32 hidden units and ReLU activations.

For the second problem (i.e. the support for updates), ASLM maintains with each model both a data array and a buffer. An insertion affects the data array only if the error of the inserted key is less than the average error of the model. Otherwise, the key is inserted into the buffer.[4] Once the buffer is full, it is merged with the data array and the corresponding model is retrained. A deletion either removes a key from the buffer, or it marks the key as deleted in the data array. In this latter case, no retraining is necessary since the other keys are not moved from their positions, and hence the performance of the current model remains unchanged.

To avoid too large (or too small) data arrays, ASLM performs a split (or merge) strategy that creates a new model (or fuses two consecutive models) once some lower (or upper) threshold on the number of elements is hit. Clearly, these split/merge operations trigger the retraining of the affected model, but this retraining is likely to be quick since the weights of the original model can be used as the starting weights for the new model.

2.2.2 Hybrid-O

The authors of RMI suggested to replace a last-stage model by a B-tree if its error exceeds a user-defined threshold. Other authors [30] suggested instead to detect the keys that cause large errors (called "outliers"), store them in a B-tree and retrain the

[4] The authors suggest organising the buffer "like a hash table". Even though hashing simplifies buffer modifications, we lose the ability to do predecessor or range queries efficiently.

model on the non-outlier keys. Among the strategies proposed to detect outliers, the most effective one (in terms of reduction of the prediction error) is to compute the average μ and the standard deviation σ of the errors produced by a trained model and to collect as outliers the keys whose prediction error falls outside $[\mu - \sigma, \mu + \sigma]$.

For the index structure, that we name Hybrid-O, the authors of [30] have proposed a single-stage index of linear models in conjunction with one B-tree for the outliers. The construction algorithm partitions the sorted dataset D into blocks of a user-given size and trains one model for each partition. Then, it collects the outliers with the strategy described above, adds them to the B-tree and finally retrains each model on the corresponding outliers-free partition. Both the training and the outliers-detection steps cost $O(n)$ time, thus making the overall time to construct Hybrid-O equal to $O(n)$, if we assume that the time to bulk-insert the outliers in the B-tree is negligible.

At query time, a first binary search determines the model responsible for estimating the position *pos* of a query key x, which is the rightmost model whose first covered key is less than or equal to x. Let j be the model index, and let min_j and max_j be its errors precomputed using (1). Another binary search for x is executed within $A[pos - min_j, pos + max_j]$ and, if the search is unsuccessful, x is searched within the B-tree associated with this model.

When inserting a new key y, if Hybrid-O detects that y is an outlier (because its prediction error falls outside $[\mu - \sigma, \mu + \sigma]$), then y is inserted in the B-tree. Otherwise, y is added to the partition that corresponds to the model responsible for y, and the model is retrained. Deletions in Hybrid-O are not discussed in [30].

2.2.3 ALEX

The Adaptive LEarned indeX (ALEX) [10] introduces a top-down adaptive way of constructing a tree-shaped linear learned index. Specifically, the linear model in the root is trained on the entire dataset D as in RMI. Then, in order to create the children of the root node, its codomain $[0, n)$ is divided into a number p (to be tuned beforehand) of equal-sized partitions which are scanned in sorted order. If the root model maps more than a user-given number m of keys into one partition, a new inner node for this partition is created, and the construction algorithm is run recursively on this node. Otherwise, the partition is merged with the subsequent partitions (as long as the number of accumulated keys is less than m), and a leaf node is created and trained on it. Note that in no case a trained model is discarded: it either becomes part of a leaf node approximating the position of the query key, or it becomes a routing model that forwards the query key to one of its children, as we will see shortly. If we assume that the partitions are balanced in terms of the number of keys which fall in each of them, then this construction process obeys the recurrence $T(n) = p\,T(n/p) + \Theta(n)$, which has solution $O(n \log n)$ for any $p > 1$.

In ALEX, each inner node stores both its learned linear model and one pointer for each children node which is responsible for a partition of the space managed by the current node. This allows to traverse the hierarchy of models without any search in-between levels by taking the currently visited node prediction, say $c(x)$,

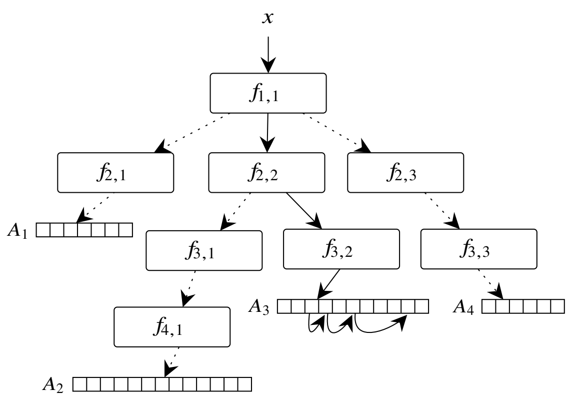

Fig. 5 To overcome the static nature of RMI, ALEX instantiates linear models as needed, growing deeper until each leaf model has approximately the same number of keys. A search is performed similarly to RMI, with the difference that here a leaf model j stores a subarray A_j of the original sorted input array A.

and following the $\lfloor p \cdot c(x)/n \rfloor$-th child pointer. Instead, a leaf node j stores the model f_j and stores the n_j keys the model was trained on in a sorted array A_j, which is a subarray of A. Once a search arrives at a leaf node j, ALEX performs an exponential search in A_j starting from the position $\lfloor f_j(x) \rfloor$. An example of a search in ALEX is depicted with solid lines in Figure 5, where for simplicity we did not show the arrays of pointers inside inner nodes.

To accommodate insertions, ALEX can be configured to store the keys at the leaves in either a Gapped Array (GA) [2] or in a Packed Memory Array (PMA) [3]. Both GA and PMA are sorted arrays that evenly intersperse empty spaces among the n_j keys so that making space for a new key requires pushing to the left or to the right of the insertion position only a small number of elements, i.e. only $O(\log n_j)$ moves for random insertion patterns. However, at the expense of a more involved algorithm, PMA guarantees $O(\log^2 n_j)$ worst-case time inserts for any insertion pattern, which is better than the $O(n_j)$ time of a GA. In both choices of the key array, ALEX uses the model to predict the approximate insertion position, which is then corrected by an exponential search.

If the distribution of keys changes after several inserts, then some leaves will become overloaded with data. In this case, the authors suggest to transform an overloaded leaf to an inner node and to create a number of child nodes, similar to what happens in the construction of the entire ALEX data structure. The net consequence is that ALEX grows deeper and deeper, possibly slowing down queries until a full index reconstruction is performed.

Deletions in ALEX are not discussed in [10], but we can observe that, after locating the key to delete with the search algorithm described above, one can use the PMA deletion algorithm that takes $O(\log^2 n_j)$ amortised time [3].

2.2.4 AIDEL

Similar to ASLM and ALEX, the learned index based on Adaptive InDEpendent Linear regression models (AIDEL) [22] addresses the poor key distribution strategy of RMI and its lack of support for updates. But, unlike the others, AIDEL provides some latency guarantees too. They hinge upon a user-given integer parameter $\varepsilon \geq 1$ indicating the maximum tolerable error for a model, and thus it proposes a construction algorithm that finds the proper number of models to match this error bound.

The construction algorithm trains a linear model on increasingly larger partitions of the key-sorted dataset, starting from the first key. A partition expands by a constant number k of keys set beforehand. At each expansion, the current model errors min_j and max_j are computed using (1). As soon as one of the two errors exceeds ε, the partition is iteratively decreased in size by a user-given fraction of k until the linear model error is not larger than ε. The last trained linear model is then appended to the result, and the process is continued on the rest of the dataset. Once the whole dataset is processed, each trained model f_j (one for each partition) is stored in a directory structure alongside the first key it covers and the values min_j and max_j.

The construction of AIDEL takes $O(n^2)$ time, because we have $\Theta(n)$ expansions, each taking $O(n)$ time to train and evaluate the error incurred by the current model.

At query time, AIDEL picks from the directory structure the model that can best approximate the position of the query key x, i.e. the rightmost f_j in the directory whose first covered key is less than or equal to x.[5] Then, a final binary search is performed on $A[pos - min_j, pos + max_j]$, where $pos = f_j(x)$. Note that this last step costs $O(\log \varepsilon)$ time due to the threshold guaranteed on min_j and max_j. It goes without saying that RMI, ASLM, Hybrid-O and ALEX have no such guarantee in the search time which may be, thus, bounded only by $O(\log n)$.

For the insertions, AIDEL adopts a simple but memory-hungry approach. It allocates a sorted list for each pair of consecutive input keys x_i, x_{i+1} that accommodates the insertions of new keys falling between x_i and x_{i+1}. When a sorted list becomes too long, it is merged with A, and the construction algorithm of the previous paragraph is rerun over the newly merged keys. The directory structure is updated accordingly.

The deletion of a key y in AIDEL is not discussed in [22], but we can observe that it can be handled by removing y from the ith sorted list if y falls between two consecutive keys x_i, x_{i+1} in A, or by marking y as deleted (e.g. with an array of flags of the same size of A) if y belongs to A.

Intermezzo: avoiding the model retraining

Some techniques have been suggested to avoid the model retraining after updates [14]. The main idea is to keep the trained model (e.g. the whole RMI hierarchy or a single model from ASLM or Hybrid-O) as is, to mark the deletions with a flag, and to correct the drift in position estimates caused by the insertions. This correction can be done in two ways.

[5] The authors do not discuss the details of this step, but we assume that it is performed in $O(\log \tilde{m})$ time via a binary search on the $\tilde{m}$ sorted key-model pairs.

The first way is to learn the update distribution, say with a CDF model G, so that the position of a key x in the index updated after n_i insertions is approximated by $nF(x) + n_i G(x)$, where F is the CDF model trained on D.

The second way is to estimate the drift by considering the (known) drift of some reference points. Specifically, the data array is divided into blocks of fixed size 2^b, and the first key inside each block is chosen as the reference point. For each reference point, we store an integer counting how many keys have been inserted in the block that precedes it. Therefore, computing the number of keys inserted before a reference point amounts to compute a prefix sum of the counters of all the blocks that precede the reference point. After the insertions, the drift of the ith reference point is computed by subtracting its true position (equal to $i2^b$ plus the number of keys inserted before it) from the position estimated by F for the reference point. At query time, the position for a key x is computed as $nF(x) - \Delta(x)$, where $\Delta(x)$ is the correction computed by interpolating the two nearest reference point drifts at the left and at the right of x.

2.3 The FITing-tree

The FITing-tree [13] has introduced a more principled way of learning the data distribution $D = \{(x, rank(x))\}_{x \in A}$. This problem is reduced to the problem of computing a Piecewise Linear Approximation (PLA) of D which guarantees a user-given maximum error $\varepsilon \geq 1$ and consists of the least amount of linear models. The first condition guarantees an upper bound on the search time, the second condition minimises the space occupancy of a learned index.

Formally speaking, the smallest PLA for a subarray $A[0, b]$ which incurs error ε has size (i.e. number of linear models)

$$T[b] = 1 + \min_{a \in C_\varepsilon(b)} T[a-1] \quad \text{with } T[0] = 1, \tag{2}$$

where $C_\varepsilon(b)$ is the set of starting points of linear models having maximum error ε and ending in $A[b]$. In other words, the minimum-size PLA for $A[0, b]$ is found by considering the minimum-size PLA for $A[0, a-1]$ along with the single linear model which covers the remaining subarray $A[a, b]$, where a is varied in such a way that the linear model covering $A[a, b]$ has a maximum error of ε.

Equation 2 leads to a dynamic programming algorithm which, however, has a prohibitive $O(n^3)$ running time. For this reason, the authors of [13] proposed a greedy linear-time algorithm that takes the first key of A, as the starting point, and then attempts to maximise the length of the linear model whose error is smaller than ε. This maximisation is done by keeping a cone defined by the origin $(A[0], 0)$, initial high slope $+\infty$, and initial low slope 0. Each time the length of the linear model is extended with a new key, the cone either narrows (decreasing one or both slopes), or stays the same. When a key $A[i]$ falls outside the cone, the process is stopped and a linear model is created for the subarray $A[0, i-1]$ taking as slope any value in the

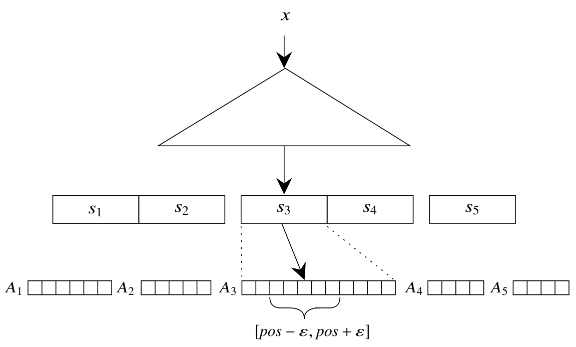

Fig. 6 A FITing-tree partitions A into subarrays A_j, each of which is indexed by a linear model with maximum error ε. The segments s_j (tuples containing the first key covered by the linear model and the parameters of the model) are indexed by a B$^+$-tree which stores in its leaf level the first keys of all those $\tilde{m}$ segments.

current slope range and using $(A[i], i)$ as the origin of a new cone. Eventually, the whole dataset is processed and the final result is a partition of A into $\tilde{m}$ variable-sized ranges that can be approximated with linear models guaranteeing maximum error ε.

In order to dig into the technical details of the FITing-tree, let us define a *segment* as a tuple containing the first key covered by a linear model (which we call the key of the segment), the parameters of the linear model (hence, its slope and intercept), and a pointer to the subarray of A containing the range of keys covered by that segment. The FITing-tree is then constructed by *indexing* via a B$^+$-tree the (first) key of each one the $\tilde{m}$ segments produced by the greedy algorithm. This way, that B$^+$-tree occupies $\Theta(\tilde{m}/B)$ disk pages and has depth $\Theta(\log_B \tilde{m})$.

At query time, the B$^+$-tree is first searched to find the segment s_j that the query key x belongs to. Then, the segment is used to predict the approximate position *pos* of the query key x inside the pointed data array A_j. Finally, a binary search is performed in $A_j[pos - \varepsilon, pos + \varepsilon]$ to find the correct position of x in the whole array A, as depicted in Figure 6. Finding the segment s_j costs $O(\log_B \tilde{m})$ I/Os, while the final binary search step costs $O(\log(\varepsilon/B))$ I/Os, because s_j is guaranteed to incur an error ε in estimating the position of x in A_j. The total cost of a query is thus $O(\log_B \tilde{m} + \log(\varepsilon/B))$ I/Os.

For the insertions, a FITing-tree adopts one of the following two strategies: in-place inserts or delta inserts.

A FITing-tree configured for in-place inserts avoids invalidating a segment by introducing a positive integer parameter β called "insert budget", by constructing a FITing-tree with maximum error $\varepsilon + \beta$, and by padding A_j with β empty slots at the beginning and at the end of it. Inserting a new key x into A_j is implemented by locating the insertion position of x and by making room for x by shifting the existing keys towards the start or the end of A_j, depending on which is closer. However, once all empty spaces in A_j are filled, the greedy algorithm must reprocess the keys in A_j. This step either allocates a larger array or it introduces new segments if one segment is not enough to guarantee an error $\varepsilon + \beta$. In this latter case, the old segment is deleted and the new ones are added to the B$^+$-tree. The disadvantage of in-place inserts is that the size of A_j can grow arbitrarily large if the input dataset shows long linear trends, thus making high the cost of shifting keys in A_j.

A FITing-tree configured for delta inserts allocates for the j-th segment a fixed-size sorted buffer. Once the buffer is full, it is merged with A_j and it is re-processed by the greedy algorithm. As before, this can either produce a single segment, with a new empty buffer, or multiple segments. The latter case requires the modification of the B$^+$-tree. The time complexity of an insertion is now linear in the size of the fixed-size buffer rather than the size of A_j, except for when the buffer is full and must be merged with A_j.

Deletions in the FITing-tree are not discussed in [13].

2.4 The PGM-index

The Piecewise Geometric Model index (PGM-index) [11, 32] improves the FITing-tree in several issues: (i) it employs a linear-time PLA construction algorithm which finds the minimum number of segments covering D with an error of at most ε; (ii) it builds upon a recursive structure that fully exploits the space/time-efficient routing of segments, thus resulting much more succinct than B$^+$-trees; (iii) it further reduces its space occupancy by means of novel techniques that compress the linear models of the optimal-sized PLA; and, finally, (iv) its structure is flexible enough that it can adapt not only to the distribution of the input keys but also to the distribution of the queried keys. The rest of this section will dig into some more details about these features.

In Section 2.3 we have seen how the problem of computing PLAs can be solved either optimally via dynamic programming in $O(n^3)$ time, or heuristically in $O(n)$ time but renouncing to optimality. Interestingly enough, this problem has been extensively studied in computational geometry, and it admits streaming algorithms which produce the minimal number of segments $m^\star$ in only $O(n)$ time [28]. The PGM-index deploys one of these optimal algorithms and stores at its lowest level the minimum-size sequence of $m^\star$ segments as triples, each consisting of the first key covered by the segment, its slope and intercept. The improvement induced by the storage of $m^\star$, instead of $\tilde{m}$, segments is significant and up to 63% [11].

The second feature of the PGM-index is its recursive structure. Its construction starts with the whole input array A that is turned into the set of two-dimensional points $D = \{(x, rank(x))\}_{x \in A}$, as we commented in the previous pages. D is processed by the optimal algorithm of [28] which produces the minimum-size sequence of $m^\star$ segments covering the keys in A with an error of at most ε. Then, the first key covered by each segment is extracted and used to form a new set of $m^\star$ keys, over which the minimum-size PLA construction algorithm is applied again. Recursion proceeds until only one segment is obtained. Each level of the recursion corresponds to a level of the PGM-index, starting from the bottom, in which there is a one-to-one correspondence between segments and nodes of the PGM-index. In a way, this approach constructs a sort of multiway search tree but with three main advantages with respect to the B$^+$-tree constructed by the FITing-tree: (i) the nodes of the PGM-index have variable fan-out driven by the (typically large) number of keys

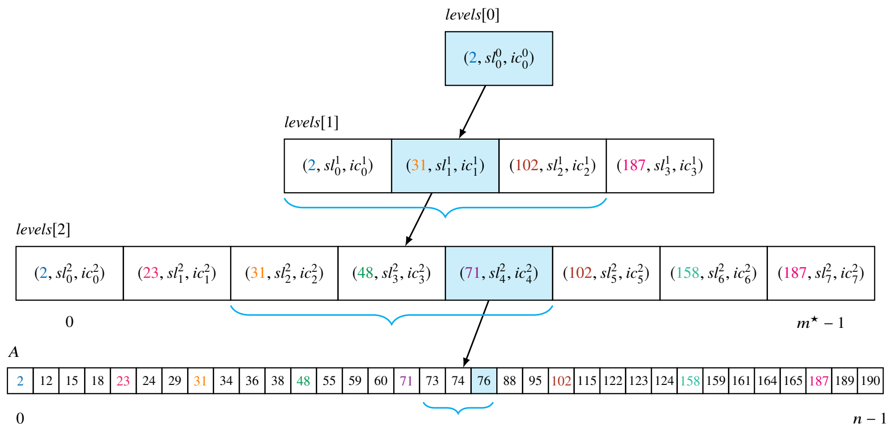

Fig. 7 Each segment in a PGM-INDEX routes the queried key to one of the segments of the level below by computing a position that is at most ε away from the correct one. In this picture $\varepsilon = 1$, the cyan nodes are the ones traversed by the search for the key $x = 76$, and the brackets specify the range where the binary search is performed: actually, the binary search is executed over the first key of each segment in the range, which is stored as the first component of the triple denoting a segment.

covered by segments, so that the height and space occupancy of the index is very small in practice; (ii) the segment associated with 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 logarithmic on ε.

At query time, the PGM-INDEX uses the current segment (initially the root) to estimate the position of the searched key among the keys of the next lower level. 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 lower level is the first key covered by a segment on that level, the binary search has identified the next segment to query. So the process continues until the last level is reached and the position in A of the queried key is determined.

Example: search in a PGM-INDEX

Consider the PGM-INDEX with $\varepsilon = 1$ in Figure 7. A query for the key $x = 76$ starts from the root node which stores the segment $levels[0][0] = (2, sl_0^0, ic_0^0)$ where 2 is its first covered key, sl_0^0 is the slope of the segment and ic_0^0 is its intercept.

Level 1. The current segment allows to compute the approximate position of the searched key $x = 76$ as $\lfloor x \cdot sl_0^0 + ic_0^0 \rfloor = 1$. Since the current segment has a maximum error of $\varepsilon = 1$, a binary search over the (first covered) keys in $levels[1][1-\varepsilon, 1+\varepsilon] = [2, 31, 102]$ suffices to determine that the correct position of x is between 31 and 102, so that the next segment responsible for x is $levels[1][1] = (31, sl_1^1, ic_1^1)$.

Level 2. The algorithm above is repeated on the current segment, by computing the approximate position $\lfloor x \cdot sl_1^1 + ic_1^1 \rfloor = 3$ which is then corrected by performing a binary search in $levels[2][3 - \varepsilon, 3 + \varepsilon] = [31, 48, 71]$. This finds that the correct position of x is after 71, so that the next segment responsible for x is $levels[2][4] = (71, sl_4^2, ic_4^2)$.

Level 3. The last level consists of the array A, and the current segment allows to compute the approximate position $\lfloor x \cdot sl_4^2 + ic_4^2 \rfloor = 17$ which is then corrected by performing a binary search in $A[17 - \varepsilon, 17 + \varepsilon] = [73, 74, 76]$. This search is successful and finds that x occurs in position 18.

It can be shown that the construction of the PGM-index reduces the number of segments at each level by a multiplicative factor larger than 2ε, so that the final data structure has $O(\log_\varepsilon m^\star)$ levels.

Theorem 1 ([11]) *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 taking $\Theta(m^\star)$ space and answers rank, membership and predecessor queries in $O(\log m^\star + \log \varepsilon)$ time and $O((\log_\varepsilon m^\star)\log(\varepsilon/B))$ I/Os, where $m^\star$ is the minimum number of segments covering A with a maximum error of ε, and B is the block size of the external memory model. Range queries are answered in an additional $O(K)$ time and $O(K/B)$ I/Os, where K is the number of keys satisfying the range query.*

The PGM-index can match the optimal worst-case query complexity of the B$^+$-tree by choosing $\varepsilon = \Theta(B)$, thus resulting $O(\log_B m^\star) = O(\log_B n)$. In practice, the fan-out of every node of the PGM-index is much larger than by 2ε as well as the value of $m^\star$ is orders of magnitude smaller than n, so that its query time and space occupancy result very small for real-world datasets. Therefore, the PGM-index can be considered the first learned drop-in replacement for traditional indexes to date.

The PGM-index can also be generalised to include nonlinear models. Indeed, there exists an $O(n \log n)$ time greedy algorithm, described in [32, §2.2], that still guarantees a maximum error ε when using nonlinear models. Preliminary experiments with shallow neural networks showed a reduction of the number of models in the lowest level of a PGM-index, but this did not reduce the overall space occupancy, suggesting that trading-off model complexity with space occupancy is an open issue that deserves further research.

For what concerns inserts, if new keys are appended to the end of the array A while maintaining the sorted order (as it occurs in time series), the update of the PGM-index is easy and efficient. In fact, the last segment can be updated in $O(1)$ amortised time and, 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 and the insertion of k is repeated recursively in the last segment of the level above. The recursion stops when a segment at any level covers k within the ε guarantee, or when the root segment is reached possibly determining its split and, thus, the creation of a new level/segment above. Since the updates at each level take constant amortised I/Os, the overall amortised I/O complexity of this insertion algorithm is $O(\log_\varepsilon m^\star)$.

For general inserts, the PGM-INDEX defines $b = \Theta(\log n)$ static PGM-INDEXes built over sets $S_0, \ldots, S_b$ of keys which are either empty or have size $2^0, 2^1, \ldots, 2^b$. Each insert of a key k, finds the first set S_i which is empty and builds a new PGM-INDEX over the set $S_0 \cup \cdots \cup S_{i-1} \cup \{k\}$. This union can be computed in linear time because we can assume that the sets S_js are sorted, and thus a simple merging creates the new sorted set which consists of 2^i keys (the sets S_js preceding S_i are full). The new merged set can be used as S_i, and the previous sets can be emptied. This algorithm can be shown to take $O(\log n)$ amortised time, while membership and predecessor queries take $O(\log n\ (\log m^\star + \log \varepsilon))$ time because every search must be executed on all the $b = \Theta(\log n)$ static PGM-INDEXes.

The analysis in the external memory model (that we omit here), completes the proof of the following result.

Theorem 2 ([11]) *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 $O(\log n\ (\log m^\star + \log \varepsilon))$ time, insertions and deletions in $O(\log n)$ amortised time. In the external memory model with block size B, membership and predecessor queries take $O((\log_B n)(\log_\varepsilon m^\star))$ I/Os, insertions and deletions take $O(\log_B n)$ amortised I/Os.*

For the results about the compression of the parameters of the segments (intercept and slopes), and the adaptability of the PGM-INDEX to the query distribution, we refer the reader to [11] and mention that an implementation of the PGM-INDEX is publicly available at https://pgm.unipi.it.

2.5 Learned multidimensional and secondary indexes

All learned indexes we discussed so far can easily be extended to handle multidimensional data such as geographic coordinates. In fact, multidimensional keys can be mapped onto one single dimension via space-filling curves that preserve spatial proximity, and then use any one-dimensional index structure over the projected points [12]. One such mapping, called Z-order or Morton order, simply concatenates the result of interleaving the binary representations of the coordinate values. For example, the three-dimensional point $(4, 7, 1) = (100, 111, 001)$ is mapped to 011 010 110 = 214. Experiments on the combination of Z-order and RMI showed that, with respect to an R-tree, space is reduced by up to 97% and the query time is improved by up to 2.5× [34].

In databases, it is common to build separate indexes on different columns of a table. In such cases, at most one index can have the same ordering of the records in the table, which is typically the index on the primary key of the table. The other indexes, called secondary and unclustered indexes, cannot. Even though we described learned indexes assuming the data being sorted by key, it is still possible to use them as secondary indexes. In fact, we can create (*value*,*pointer*) pairs where each value in the indexed column is associated with a pointer to the original record (or a list of pointers in the case of non-unique values). A pointer can be a record identifier or the primary key of the record. Then, we sort the pairs by value and create a learned secondary index on it. At query time, the learned secondary index locates the (*value*,*pointer*) pair associated with the sought *value* and outputs the record pointed by *pointer*. Note that the space overhead of using this kind of indirection for secondary indexes occurs also if one uses traditional indexes, but learned indexes have the additional benefit of being more succinct in space and possibly faster in query time.

An alternative approach to build secondary indexes consists of capturing the correlation between the target column T and a host column H for which there already exists an index. The Tiered Regression Search Tree (TRS-TREE) [35] learns the mapping $H = h(T)$ by recursively dividing T's value range into a number of equal-sized subranges until every pair (t, h) of values from T and H covered by the corresponding subrange can be well estimated using linear regression. Specifically, the construction starts from the root node being the current node and associates with each node a range of T, which for the root is the whole range of T's values. The algorithm retrieves from the table all pairs (t, h) such that t is within the range associated with the current node and trains a linear regression model on such pairs. Then, the pairs on which the linear model makes an error greater than a user-given parameter ε are inserted into an outlier buffer. If the size of the outlier buffer exceeds a fixed amount, the current range is divided into a fixed number p of equal-sized subranges, the current node becomes an internal (routing) node, and the construction process is repeated recursively on the p children of the current node, one for each subrange. Otherwise, the node becomes a leaf node and it stores the linear regression model parameters and the outlier buffer, which is implemented as a hash table mapping from t to the corresponding record identifier (either a primary key or a tuple location).

At query time, a predicate $P = lb \leq T \leq ub$ is implemented via a breadth-first search that starts from the root node and visits all children whose range overlaps with P. At a leaf node with range r, the endpoints of the intersection between P and r are mapped to a range on H via the linear model stored in the leaf, and the outliers are collected from the buffer. Once the breadth-first search is completed, the ranges on H are used to query the existing index on the host column, while the outliers are fetched directly by visiting the corresponding record identifiers. Since the TRS-TREE may return false positives, the fetched tuples are compared against P and possibly filtered out from the result of the query.

2.6 Comparison among learned indexes

A comparison among the learned indexes discussed so far is given in Table 2, where we use the following terminology and notation. First of all, the complexity bounds are given in terms of the number n of keys in the input dataset, the number ℓ of levels in the DAG structure of RMI, the (sub-optimal) number $\tilde{m}$ of segments which cover the keys in the input dataset with error ε, and the optimal number $m^\star$ of segments which cover the keys in the input dataset with error ε, as computed by the algorithm of [28].

The "Structure" column groups learned indexes according to how they arrange the (learned and non-learned) models they are composed of. The most common kind of structure is a *tree*, which partitions the input array A so that each model specialises on a specific subarray of A that gets smaller and smaller as the depth of the model in the structure grows. In a tree-structured learned index, a query always follows a root-to-leaf path, where the next model is either chosen via a fixed number of key comparisons (as in FITing-tree or PGM-index) or via a model prediction (as in RMI and ALEX). Differently, we say that the structure is a *Direct Acyclic Graph* (DAG) when a model can have more than one parent model: this is the case of an RMI with three or more stages. Lastly, we mention that some learned indexes have a *flat* structure, in which there is only one level of learned models and a comparison-based search algorithm, such as linear or binary search, that corrects the prediction error incurred by the learned models. A flat structure is convenient only when the models

Table 2 Comparison among known learned indexes. The integer $\varepsilon \geq 1$ denotes the bound on the prediction error, $\tilde{m}$ represents the number of linear models computed by the greedy algorithms in [13, 22], while $m^\star \leq \tilde{m}$ represents the minimum number of linear models computed by the optimal algorithm [28].

Name	Structure	Complexities			Properties			
		Constr. time	Query time	Query I/Os	Adaptive structure	Bounds the error	Optimal space	Insertions/Deletions
RMI [21]	DAG	$O(\ell n)^\dagger$	?	?	–	–	–	–
ASLM [23]	Flat	$O(n)^\dagger$	?	?	◐	–	–	●
Hybrid-O [30]	Flat	$O(n)$	?	?	◐	–	–	●
ALEX [10]	Tree-like	$O(n \log n)$	?	?	◐	–	–	●
AIDEL [22]	Flat	$O(n^2)$	$O(\log \tilde{m})^\S$	$O(\log(\tilde{m}/B))$	●	●	–	●
FITing-tree [13]	Tree-like	$O(n)$	$O(\log \tilde{m})^\S$	$O(\log_B \tilde{m})^\ddagger$	●	●	–	●
PGM-index [11]	Tree-like	$O(n)$	$O(\log m^\star)^\S$	$O(\log_\varepsilon m^\star)^\ddagger$	●	●	●	●

● = provides property; ◐ = partially provides property; – = does not provide property; [†]assuming models with linear time training; [‡]$\varepsilon = \Omega(B)$; [§]plus $O(\log \varepsilon)$.

can be kept in the faster levels of the memory hierarchy; otherwise, it is better to keep the models in a B$^+$-tree (as the FITing-tree does) or to recursively build a smaller/faster routing structure composed of levels of learned models (as proposed by the PGM-index).

The “Adaptive structure” column shows to what extent a learned index automatically allocates the proper kind and number of models, thus tuning itself to the best configuration for the input dataset. This feature is especially important in scenarios where (i) the software engineer does not have a deep understanding of the learned index and its possible models, for example, because the index is part of a larger software system; (ii) the tuning process is not feasible, such as in resource/time-constrained devices/applications; or (iii) the data distribution changes over time, such as when new batches of data trigger a rebuild of the index to avoid performance degradation.

The “Bounds the error” column shows which learned indexes allow the user to control the maximum error incurred by the learned models. This, in turn, guarantees an upper bound on the search latency. A learned index that does not have such property can be susceptible to unpredictable query times, especially when data is too large to be stored in the faster levels of the memory hierarchy.

The last column shows which learned indexes support *insertions and deletions* without a full reconstruction of the data structure.

3 Learned data structures for exact membership

There are applications which do not demand sophisticated operations like rank, predecessor or range queries. Dropping them and focusing only on membership queries can substantially simplify the design of data structures and improve the time complexity of the queries, as we will see shortly.

The *dictionary problem* asks to store a set S of n keys drawn from an universe $\mathcal{U}$ in order to support membership queries, insertions and deletions of keys. A key in S can also be associated with auxiliary data, in which case we aim also to support the lookup operation which, given a query x, returns the data associated with x if $x \in S$, or nil otherwise.

The historical solution to the dictionary problem is provided by hash tables. Hash tables are composed of an array T of size $m \geq n$, a hash function $h\colon \mathcal{U} \to \{0, \dots, m-1\}$ that assigns each key in the universe to a location in T, and a strategy to handle the collisions that occur when more than one key maps to the same location. Some examples of these strategies are (see [25] for a complete discussion): chaining, where $T[h(x)]$ is the head of a linked list containing elements having the same hash value $h(x)$, thus allowing $O(1 + n/m)$ expected time membership queries; or Cuckoo hashing, which uses two hash functions h_1, h_2 and places an element x in either $T[h_1(x)]$ or $T[h_2(x)]$, thus allowing $O(1)$ worst-case time membership queries.

In Section 2, we have seen how the indexing problem boils down to learn the CDF model $F(x)$ of the input set S of keys. It turns out that the same function F can be used to implement $h(x)$ as $\lfloor mF(x) \rfloor$. If the model F perfectly learns the empirical CDF of S and m is sufficiently large, then no conflicts would occur and thus the membership query could be solved in constant worst-case time. In general, the final performance depends on which hash table architecture is used (e.g. table size, collision resolution), how well F approximates the true empirical CDF of S, and how much F is difficult to compute. For example, using several datasets and a table of size $m = n$, RMI with 100K linear models in the second stage reduced the number of conflicts by about 45% with respect to a MurmurHash function [21], but at the cost of increased time complexity. In fact, for 8-byte keys with 12-bytes auxiliary data, and a hash table with chaining having size m varied from 75% to 125% of n, the lookup times of using the learned hash function with respect to MurmurHash increased on average by 40%.

Very recently, Pavo [36] suggested an alternative approach for string queries. Pavo implements h via a rather complicated graph structure with recurrent neural networks as nodes. Specifically, the input strings are divided into bigrams and fed first to a disperse step, which is an RMI-like hierarchy of experts whose purpose is to split the dataset into subsets that can be learned more easily. Each model in this disperse step is a recurrent neural network trained to “predict the MurmurHash” of a given key.[6] Then, a mapping step uses an unsupervised approach to evenly distribute inputs of the previous step to a range of T. Experimentally, Pavo was shown to reduce the average chain lengths by up to 50%, but it increased by 3–4 times the cost of computing the hash value with respect to MurmurHash.

Discussion. At the moment of writing, it seems unlikely that learned hash tables can compete with traditional and well-established hash tables. In fact, in contrast to traditional hash functions, the learned approaches described above increase the construction time due to the training step, the number of accesses in memory to retrieve the parameters of the model and the time to compute the hash due to the matrix multiplications required for model inference. As a matter of fact, if we are willing to spend more space (and time) for a learned hash function only to reduce the number of collisions, then we might as well increase the size of the hash table itself and keep using a fast traditional hash function. More importantly, a CDF model F used as a hash function of the form $h(x) = \lfloor mF(x) \rfloor$ does not distribute keys uniformly in the hash table, and this could be exploited by an adversary to create arbitrarily long collision chains.

[6] We remark that MurmurHash is a *known* function which can be implemented in a few dozen lines of C code.

4 Learned data structures for approximate membership

There are situations in which the universe $\mathcal{U}$ of keys is very large, and thus the keys are long enough to take a lot of space to be stored in a dictionary structure like the ones described in Section 3. As an example, a web crawler could use a hash table to filter out the web pages it has already visited, but the long URLs would quickly consume all the available memory. In these cases, we can slightly relax the definition of membership query, introduce some errors in the returned answers, and eventually use much more space-efficient structures.

The *approximate membership problem* asks to store a set S of n keys drawn from a universe $\mathcal{U}$ and support membership queries by admitting some (controlled) errors in the answer. More precisely, for a query on the key $x \in S$, it is reported that $x \in S$ (no false negatives). But for a query on the key $x \notin S$, it is reported erroneously that $x \in S$ (a false positive) with probability at most ϵ.

The most famous and classic data structure for this problem is the Bloom filter [5]. A Bloom filter consists of an array of m bits, initially all set to $\mathtt{0}$, and k independent hash functions $h_1, \dots, h_k$ with codomain $\{1, \dots, m\}$. For each $x \in S$, the bits $h_i(x)$ are set to 1 for $1 \leq i \leq k$. A query for y is answered affirmatively if all bits at $h_i(y)$ are 1 and negatively otherwise, as shown in Figure 8. The false positive probability of a Bloom filter is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \approx \left(1 - e^{-kn/m}\right)^k. \tag{3}$$

Given m and n, the optimal number of hash functions is $k = (m/n) \ln 2$, which gives a false positive probability of about $(0.6185)^{m/n}$.

Once again, ML has given us a different perspective on this classic algorithmic problem. Indeed, the approximate membership problem can be framed as a supervised binary classification task in which the dataset $D = \{(x, 1)\}_{x \in S} \cup \{(x, \mathtt{0})\}_{x \in \mathcal{U} \setminus S}$ is

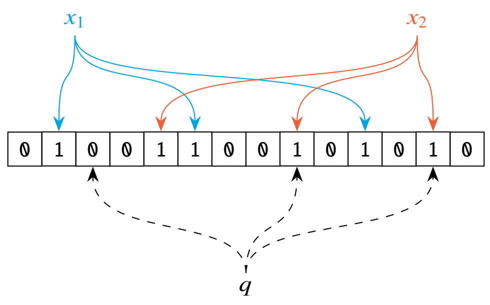

Fig. 8 The membership query q in a Bloom filter with two elements x_1 and x_2 is answered negatively because one of the three hash functions maps q to a cell containing $\mathtt{0}$.

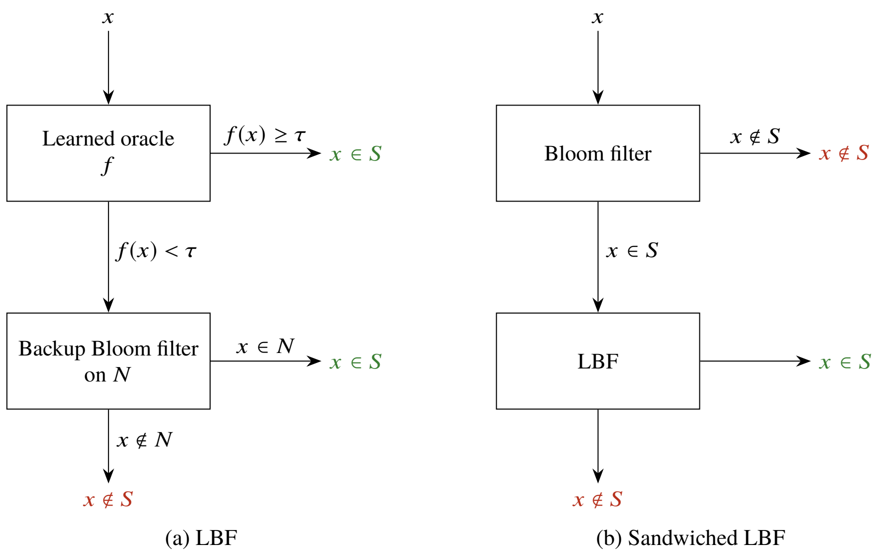

Fig. 9 (a) A Learned Bloom Filter (LBF) uses a learned oracle f to classify the membership of x, and it uses a backup Bloom filter on the set of false negatives N that x produces on S. (b) A Sandwiched LBF adds an initial Bloom filter before the LBF.

given, and we look for a model $g\colon \mathcal{U} \to \{0, 1\}$ mapping keys from the universe to a boolean indicating whether or not a key is in S. Note that the set $\mathcal{U} \setminus S$ can be huge, and in practice one would choose a subset of it.

4.1 Learned Bloom filters

The first learned approximate membership data structure, called Learned Bloom Filter (LBF) [21], uses a neural network $f\colon \mathcal{U} \to [0, 1]$ with sigmoid activations in the output layer and is trained to minimise the log loss function

$$L = \sum_{(x,y)\in D} y \log f(x) + (1 - y) \log(1 - f(x)).$$

The output value $f(x)$ can be then interpreted as a “probability” that a given key x belongs to S. Thus, the model f can be turned into a learned classifier of membership by simply setting a threshold τ on $f(x)$, and declaring $x \in S$ iff $f(x) \geq \tau$. However, unlike classic Bloom filters, this learned approach may introduce false negatives. As a solution, the set of false negatives $N = \{x \in S \mid f(x) < \tau\}$ is used to construct a smaller “backup” classical Bloom filter B with a fixed false positive rate FPR_B, as depicted in Figure 9a.

For a given test set of missing keys $T \subseteq \mathcal{U} \setminus S$, the empirical false positive rate of the model is $\mathrm{FPR}_\tau = |\{x \in T \mid f(x) \geq \tau\}|/|T|$. Therefore, the overall empirical false positive rate of an LBF is $\mathrm{FPR}_O = \mathrm{FPR}_\tau + (1 - \mathrm{FPR}_\tau)\mathrm{FPR}_B$. With a good choice of f and a proper tuning of τ on a validation set, the LBF can obtain the desired false positive rate while reducing the space consumption with respect to a traditional Bloom filter. Indeed, experiments on 1.7 million URLs showed that LBF reduced the space of a Bloom filter with 1% false positive rate from 2.04 MB to 1.31 MB (36% less), while with 0.1% false positive rate the space reduced from 3.06 MB to 2.59 MB (15% less). Subsequent research [24] extended LBFs to take tuples as input while allowing wildcard membership queries, i.e. queries in which only a subset of tuple entries are specified.

From a theoretical perspective, it has been shown that, under the assumption that the test set T and the future queries Q have the same distribution, the empirical false positive rate of an LBF is close to the false positive rate on Q, as both are concentrated around the expectation.

Theorem 3 ([26]) *Consider a learned Bloom filter* $f\colon \mathcal{U} \to [0,1]$ *with threshold* τ *and backup Bloom filter* B*. Consider a test set* T *and a query set* Q*, where* T *and* Q *are both determined from samples according to a distribution* $\mathcal{D}$*. Let* X *be the empirical false positive rate on* T*, and* Y *be the empirical false positive rate on* Q*. Then*

$$\Pr(|X - Y| \geq \epsilon) \leq e^{-\Omega(\epsilon^2 \min(|T|,|Q|))}.$$

It is important to note here that if the assumption of Theorem 3 is not met (e.g. if the query set distribution changes over time and it is no more “close” to the one of the test set) then, unlike in the traditional Bloom filter, there are no guarantees on the number of false positives of an LBF.

4.2 Sandwiched learned Bloom filters

The Sandwiched Learned Bloom Filter (Sandwiched LBF) [26] improves the LBF by adding a Bloom filter before using f in order to remove most queries for keys not in S. At query time, the initial Bloom filter (BF) forwards to f the keys that it would declare to belong to S, but returns an immediate negative answer otherwise. Then, as before, f attempts to remove false positives and the backup filter is used to remove the false negatives (see Figure 9b). The advantage gained by adding the initial BF is that it reduces the number of false positives passed to f and, in turn, it allows the backup filter to be weaker, and thus much smaller in space. Moreover, the initial BF may make the overall learned approach more robust if the queries do not come from the same distribution of the test set used to estimate the false positive rate of the LBF.

To analyse the Sandwiched LBF on a set S of n keys, we assume a total budget of bn bits to be divided between the initial Bloom filter of $b_1 n$ bits and the backup

Bloom filter of $b_2 n$ bits. Assume also that we can pick an oracle f with a false positive probability of F^+ that produces nF^- false negatives on S. Let us model the false positive rate of a BF that uses j bits per stored key as α^j, where $\alpha \approx 0.6185$ in the standard Bloom filter. The false positive rate of a Sandwiched LBF is then

$$\alpha^{b_1}\left(F^+ + (1-F^-)\,\alpha^{b_2/F^-}\right). \tag{4}$$

Indeed, for a key $y \notin S$ to be declared as a member of the set, y first has to pass through the initial Bloom filter with probability α_1^b, and then either f (wrongly) classifies y as a member of S with probability F^+ or, with remaining probability $(1 - F^+)$, it passes y to the backup Bloom filter, whose false positive rate is α^{b_2/F^-}. Setting to zero the derivative of Equation 4 with respect to b_1 yields

$$\frac{F^+F^-}{(1-F^+)(1-F^-)} = \alpha^{(b-b_1)/F^-} = \alpha^{b_2/F^-},$$

so that the false positive rate is minimised at

$$b_2^* = F^- \log_\alpha \frac{F^+F^-}{(1-F^+)(1-F^-)}.$$

This last equation reveals that the size of the backup Bloom filter does not depend on the budgeted number of bits per key bn. Therefore, after appropriately sizing the backup Bloom filter to prevent false negatives due to f, it is better to spend the remaining budgeted bits on the initial filter to get rid of false positives up front.

Example: performance of Bloom filters, LBFs and Sandwiched LBFs

Suppose that we have a budget of $b = 8$ bits per key, and that we pick an oracle f with $F^+ = 0.01$ and $F^- = 0.5$. Let $\alpha = 0.6185$ as in the standard Bloom filter. Then

- A standard Bloom filter achieves a false positive probability of

$$\alpha^b \approx 0.0214.$$

- An LBF achieves a false positive rate of

$$F^+ + (1-F^+)\,\alpha^{b/F_-} \approx 0.0105.$$

- A Sandwiched LBF, that uses the optimal $b_2^* \approx 4.7820$, achieves a false positive rate of

$$\alpha^{(b-b_2^*)}\left(F^+ + (1-F^-)\,\alpha^{b_2^*/F^-}\right) \approx 0.0043.$$

4.3 Neural Bloom filters

A Neural Bloom filter [31] implements the learned oracle of the LBF with a memory-augmented neural network. The network learns to address a real-valued memory matrix M by classifying which memory slots to read or to write depending on the input. Specifically, the network consists of the function f_{enc}, which is a convolutional neural network in the case of image inputs or a long short-term memory network in the case of text inputs, and the functions f_w, f_q, f_{out}, which are multilayer perceptrons. The input x is encoded into an embedding $z \leftarrow f_{enc}(x)$, then transformed into a write vector $w \leftarrow f_w(z)$ and a query vector $q \leftarrow f_q(z)$. The address is computed as $a \leftarrow \mathrm{softmax}(q^\intercal A)$, where A is a learnable address matrix. A write is performed with an additive write operation to memory weighted by the address, $M \leftarrow M + wa^\intercal$. A read is performed by a component-wise multiplication of the address with the memory, $r \leftarrow M \odot a$. The read vector r is then fed along with w and z to f_{out} that produces a single scalar probability $o \leftarrow f_{out}([r, w, z])$. Similar to LBFs, the Neural Bloom filter fixes a threshold τ on the output probability and uses a backup Bloom filter to correct false negatives.

Experiments on storing 5K strings with 1% false positive rate showed that the space of Neural Bloom filters with respect Bloom filters reduced from 47.9 KB to 1.5 KB (32× less). With 0.1% false positive rate, the space reduced from 72.2 KB to 24.5 KB (3× less).[7] However, both Learned and Neural Bloom filters had uncompetitive query latencies (around 400× slower than classic Bloom filters) and throughputs (20× lower for a batch of 10K queries).

4.4 Adaptive learned Bloom filters

All the learned Bloom filters discussed so far output that a key x belongs to S when $f(x) \geq \tau$, regardless of the confidence $f(x)$ the model f has on the membership of x. As depicted in Figure 10, an Adaptive learned Bloom filter (Ada-LBF) [9] exploits this confidence to partition the elements of S into r regions delimited by the thresholds $0 = \tau_0 < \tau_1 < \cdots < \tau_{r-1} < \tau_r = 1$. A region j contains $n_j = |\{x \in S \mid \tau_{j-1} \leq f(x) < \tau_j\}|$ keys from S, which are hashed using k_j independent hash functions with codomain $\{1, \ldots, m\}$ and inserted into a shared Bloom filter of m bits. Interestingly, the LBF can be seen as a particular case of the Ada-LBF where $r = 2, \tau_1 = \tau, k_1 = k$ and $k_2 = 0$.

Similar to Bloom filters (see Equation 3), the expected false positive rate of the jth region of an Ada-LBF is

[7] It has to be noted that for 5K elements a Bloom filter of ≈ 6 KB and $k = 7$ hash functions would have sufficed to achieve 1% false positives. Similarly, a Bloom filter of ≈ 9 KB and $k = 10$ hash functions would have sufficed to achieve 0.1% false positives.

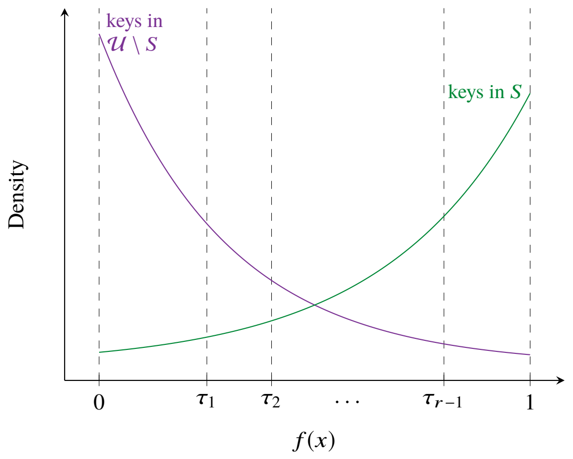

Fig. 10 Ideally, the distribution of the outputs returned by a membership oracle f, which we can interpret as the confidence of the oracle, is increasing for keys in S and decreasing for keys not in S. Ada-LBF partitions this distribution into r of regions and uses a proper number of hash functions for each region.

$$\mathbb{E}(\mathrm{FPR}_j) = \left(1 - \left(1 - \frac{1}{m}\right)^{\sum_{i=1}^{r} n_i k_i}\right)^{k_j} = \gamma^{k_j}. \tag{5}$$

Moreover, the probability that a key y sampled from a query distribution $\mathcal{D}$ over $\mathcal{U} \setminus S$ (denoted below with $y \sim \mathcal{D}$) is mapped by f to the region j can be expressed as

$$p_j = \Pr_{y \sim \mathcal{D}}(\tau_{j-1} \le f(y) < \tau_j). \tag{6}$$

Putting together Equations 5 and 6, the expected overall false positive rate of an Ada-LBF is

$$\mathbb{E}(\mathrm{FPR}) = \sum_{j=1}^{r} p_j\, \mathbb{E}(\mathrm{FPR}_j) = \sum_{j=1}^{r} p_j \gamma^{k_j}. \tag{7}$$

In practice, for a given test set of missing keys $T \subseteq \mathcal{U} \setminus S$, we can estimate p_j empirically as $\hat{p}_j = |\{x \in T \mid \tau_{j-1} \le f(x) < \tau_j\}|/|T|$. Furthermore, since (7) is hard to minimise due to the many hyperparameters (i.e. τ_j and k_j for $1 \le j \le r$), we simplify the tuning as follows:

1. Choose a number k_{max} of hash functions for the first region, i.e. $k_1 = k_{max}$.
2. Fix $k_r = 0$. In other words, if $f(x) \ge \tau_{r-1}$, we return that $x \in S$ without checking the Bloom filter, similar to what happens in an LBF (cf. Figure 9a).
3. Choose a constant $c > 1$ and fix $\hat{p}_j / \hat{p}_{j+1} = c$, $k_j - k_{j+1} = 1$ for $j = 1, 2, \ldots, r-1$. This ensures that, as j diminishes, k_j increases linearly and $\hat{p}_j$ grows exponentially fast.

This leaves us with only two parameters, k_{max} and c.

Analogously to Theorem 3 for the LBF, it is possible to show that, under the assumption that the test set T and the future queries have the same distribution, the empirical false positive rate of an Ada-LBF is close to the real $\mathbb{E}(\mathrm{FPR})$.

Theorem 4 ([9]) *Consider an adaptive learned Bloom filter built on a set S, consisting of r regions. Consider a test set T and a query set Q, where T and Q are both determined from samples according to a distribution $\mathcal{D}$. Then, $\sum_{j=1}^{r} |\hat{p}_j - p_j|$ converges to 0 in probability as $|T| \rightarrow \infty$.*

The following result shows under which conditions the Ada-LBF improves the false positive rate of the corresponding (i.e. same array size and same oracle) LBF.

Theorem 5 ([9]) *Consider a learned Bloom filter $f\colon \mathcal{U} \rightarrow [0,1]$ with threshold τ and backup Bloom filter of size m with k hash functions. Consider an adaptive learned Bloom filter with the same oracle f, m bits, r regions, $\tau_{r-1} = \tau$, γ as defined in Equation 5, and $p_j/p_{j+1} \geq c > 1$ for $j = 1, 2, \ldots, r-1$. If there exists $\lambda > 0$ such that $c\gamma \geq 1 + \lambda$ holds, $n_{j+1} - n_j > 0$ for $j = 1, 2, \ldots, r-1$, and $r \leq 2k$ is large enough, then the adaptive learned Bloom filter has a smaller false positive rate than the learned Bloom filter.*

Authors of [9] discuss also a variant called Disjoint Ada-LBF which hashes the keys falling a region to a dedicated Bloom filter rather than to the shared Bloom filter. Experimentally, on a task to identify $n \approx 80K$ malicious URLs, Ada-LBF with a random forest oracle (using input features like hostname length, path length, etc.) reduced the false positive rate by 81% compared to LBF and Sandwiched LBF still using the same oracle and size (500 Kb). In turn, Disjoint Ada-LBF reduced the false positive rate by 84%. To achieve a false positive rate of $\approx 0.35\%$, Ada-BF and Disjoint Ada-BF used 300 Kb (-40%) with respect to 500Kb of LBF and Sandwiched LBF.

Discussion. The learned approaches to the approximate membership problem seem promising. A model that classifies whether an item belongs to the set by exploiting the item's features can reduce the overall size taken by the filter.

However, before replacing Bloom filters with any of the above learned variants, it is important to understand that the guarantees offered by the latter differ significantly from the former. For example, the false positive *probability* of Bloom filters holds for any possible query set, while the false positive *rate* of their learned counterparts is highly dependent on the chosen test set. Consequently, unless the queries have the same distribution of the test set, the actual false positives of the LBF may be substantially larger than expected. For example, if the learned filter is used to prevent accesses to a slow cache, then an adversary can exploit its weakness to perform a denial-of-service attack.

5 Learned data structures for frequency estimation

In the previous sections, we focused on the storage and the retrieval of keys. There are situations in which we only want to keep and retrieve some statistics of the input data, which may be too large to store and to analyse in full efficiently. In these cases, we must content ourselves with synopses of the data, which typically use little space and are efficient in providing very accurate approximate answers [7]. A famous problem in this setting is the following one.

Consider an infinite stream of updates to an array $A = [a_1, \ldots, a_n]$ of n counters, initially set to zero, where the t-th update is a pair (i_t, c_t) indicating that the i_t-th element of A increased by the value c_t. The *frequency estimation problem* asks to build a data structure that summarises the stream and allows to estimate the current value of an element a_i, also called “count” or “frequency” of the i-th element of A.

The most widely used data structure for this problem is the Count-Min (CM) sketch [7]. The CM sketch with parameters (ϵ, δ) consists of a two-dimensional array C of counters of width w and depth d, initially set to zero, and d hash functions $h_1, \ldots, h_d \colon \{1, \ldots, n\} \to \{1, \ldots, w\}$. When an update (i, c) arrives in the stream, the CM sketch is updated by changing $C[j, h_j(i)] \leftarrow C[j, h_j(i)] + c$ for each row $1 \leq j \leq d$. At query time, the count for the i-th element of A is estimated as $\hat{a}_i = \min_{1 \leq j \leq d} C[j, h_j(i)]$, as depicted in Figure 11.

Of course, two items hashing to the same bucket affect each others’ estimate. But the approximation guarantee is that if $w = \lceil e/\epsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, the estimate $\hat{a}_i$ obeys $a_i \leq \hat{a}_i$ (one-sided error); and, with probability at least $1 - \delta$, $\hat{a}_i \leq a_i + \epsilon \|A\|_1$. Here, $\|A\|_1$ is the sum of the absolute values of A’s elements.

If items with large counts (heavy hitters) collide with other items, then some estimates provided by the CM sketch may have large errors. Even though it has been shown that skewed distributions improve the estimations of CM sketch [7], treating the items with large counts separately from the CM sketch can increase the overall estimation accuracy. For the rest of the discussion, we assume that only positive $c > 0$ updates are possible and discuss ML-based solutions to the frequency estimation problem.

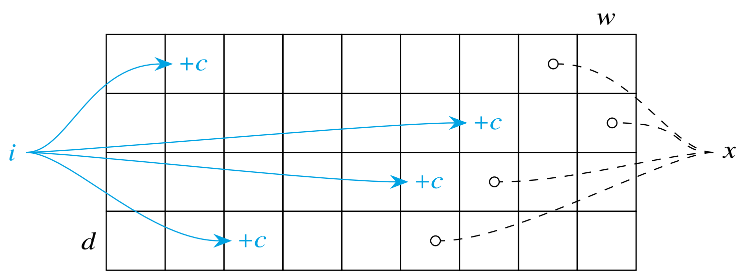

Fig. 11 In a Count-Min (CM) sketch, the hash function associated with a row maps the item i from the update (i, c) to one of the w counters in the row, which is then incremented by c. The count of an item x is estimated by taking the minimum over all the d selected counters.

5.1 ML-oracle classifying heavy hitters

The Learned CM sketch [16] trains beforehand an oracle f to determine whether an item is a heavy hitter or not. At query time, all items classified by f as heavy hitters are assigned to unique buckets storing their exact count. All the other items are forwarded to a CM sketch, as shown in the block diagram in Figure 12. Note that f should not be regarded as a lookup table, that is, it does not learn the identity of heavy hitters but the properties that allow to identify them. As an example, when applied to network traffic, an oracle from [16] based on recurrent neural networks, fed bit-by-bit with the packet source/destination IP and trained with the squared loss to predict the packet count, was able to eventually group flows with similar IP prefixes together, thus reflecting the hierarchical nature of Internet addresses.

The following theorem shows that if f has perfect accuracy and the frequency of items is from a Zipfian distribution $a_i = 1/i$, then the error of the Learned CM sketch is up to a logarithmic factor smaller than that of its non-learned counterpart.

Theorem 6 ([16]) *Let $A = [a_1, \ldots, a_n]$ be an array of n frequencies such that $a_i = 1/i$, and let $[\hat{a}_1, \ldots, \hat{a}_n]$ be the estimates of an algorithm C solving the frequency estimation problem. Define the expected error of C as $\sum_{i=1}^{n} |\hat{a}_i - a_i| a_i$. Then,*

1. *The expected error of a CM Sketch of width k and depth b/k is*

$$O\Big(\frac{k \ln n}{b} \ln^{\frac{k+2}{k-1}} \frac{kn}{b}\Big).$$

2. *The expected error of a Learned CM sketch with b_u unique buckets reserved for heavy hitters and $b - b_u$ buckets used by a CM sketch of width k and depth $(b - b_u)/k$ is*

$$O\Big(\frac{1}{b - b_u} \ln^2 \frac{n}{b_u}\Big),$$

   assuming a heavy hitter classifier with perfect accuracy.

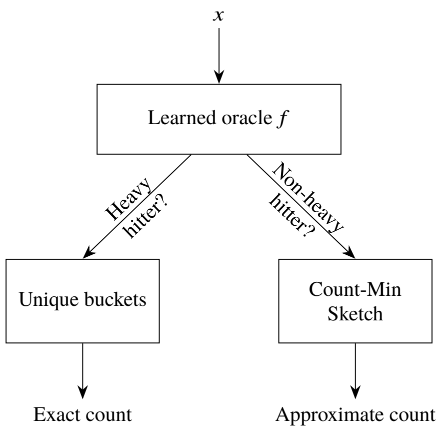

Fig. 12 A Learned CM sketch uses an ML-oracle to classify items with large counts, called "heavy hitters", and reserves for them unique exact counters.

In the network traffic scenario we mentioned earlier, the Learned CM sketch reduced the estimation error by 32% compared to CM sketch with a space of 0.5 MB and by 42% with a space of 1 MB. However, the inference time was 2.8 µs per item on a GPU, while CM sketches on CPU were shown to be four orders of magnitude faster [8].

5.2 ML-oracle estimating heavy hitters' counts

A different approach proposed in [37] uses an ML model both to classify heavy hitters and to predict the counts of the heavy hitters while maintaining the one-sided error guarantee of CM sketch. The construction algorithm takes a training set (X, Y) of items X and corresponding counts Y, sorted in ascending order by Y. It sets a boundary $P = Y[p]$ so that the first p counts of Y have an overall sum less than a given fraction $t \in (0, 1)$ of the total, i.e. $\sum_{i=1}^{p} Y[i] \leq t\|Y\|_1$. Then, the algorithm modifies the training set by increasing the distance between each count in Y and the boundary P by an offset proportional to $\epsilon\|Y\|_1$, which is the upper bound on the error of a CM sketch. Finally, an ML model f is trained on (X, Y) until (i) it does not misclassify heavy-hitters, i.e. items with a count greater than P; and (ii) it overestimates the count of heavy hitters within some desired accuracy. Once the training is finished, training items i satisfying $f(i) < P$ are stored in a traditional CM sketch. At query time, the count for an item x is estimated as $f(x)$ when $f(x) \geq P$, and it is estimated by the CM sketch otherwise.

The following theorem illustrates that, under certain assumptions, the Learned CM sketch has an upper error bound no worse than a traditional CM sketch.

Theorem 7 ([37]) *The Learned Count-Min sketch with a threshold $t \in (0, 1)$ and a backup Count-Min sketch with parameters $(\epsilon/t, \delta)$ can return the accuracy guarantee no worse than that provided by a standard Count-Min sketch with parameters (ϵ, δ), on the assumption that the items for testing the model share the same distribution with the query items, and the guarantee of the model works not only on the test set but also on the query items.*

Using a 3-layer neural network oracle with 20 hidden units, this Learned CM sketch compared to a CM sketch reduced the mean squared estimation error by about 92% on normally-distributed counts and by 73% on Zipf-distributed counts.

Authors discuss a variant called Learned Augmented Sketch that stores the counts of the top-k items into exact counters, estimates the (remaining) heavy-hitters with f, and uses the CM sketch otherwise.

Discussion. To conclude this section, we want to stress the remarks of Section 4, which apply to this problem too. That is, one should be aware that the Learned CM sketch could not be as robust as the traditional version if either the query distribution or the input stream distribution changes over time with respect to the training data. On the other hand, if these conditions hold and more efficient implementations are made available, the practical advantages of a Learned CM sketch should be taken into consideration.

6 Conclusions

This is a field still in its infancy that needs a lot of study and experiments to provide a formal framework and practical support to its preliminary, successfully promising achievements. We refer the readers to the papers in the following bibliography for a complete list of open problems, and we content ourselves here to mention that it is crucial to check also the impact of ML-based solutions in real software and to experiment with their combination with compact data structures [27], possibly in automatic engines [17].

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), and by the European Integrated Infrastructure for Social Mining and Big Data Analytics (SoBigData++, Grant Agreement # 871042).

References

1. Ao, N., Zhang, F., Wu, D., Stones, D.S., Wang, G., Liu, X., Liu, J., Lin, S.: Efficient parallel lists intersection and index compression algorithms using graphics processing units. Proceedings of the VLDB Endowment **4**(8), 470–481 (2011)
2. Bender, M.A., Farach-Colton, M., Mosteiro, M.A.: Insertion sort is $o(n \log n)$. Theory of Computing Systems **39**(3), 391–397 (2006)
3. Bender, M.A., Hu, H.: An adaptive packed-memory array. ACM Transactions on Database Systems **32**(4) (2007)
4. Bentley, J.L., Yao, A.C.C.: An almost optimal algorithm for unbounded searching. Information Processing Letters **5**(3), 82 – 87 (1976)
5. Broder, A., Mitzenmacher, M.: Network applications of Bloom filters: A survey. Internet Mathematics **1**(4), 485–509 (2004)
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press (2009)
7. Cormode, G., Garofalakis, M., Haas, P.J., Jermaine, C.: Synopses for massive data: Samples, histograms, wavelets, sketches. Foundations and Trends© in Databases **4**(1–3), 1–294 (2011)
8. Cormode, G., Muthukrishnan, S.: Summarizing and mining skewed data streams. In: Proceedings of the International Conference on Data Mining, SDM, pp. 44–55. SIAM (2005)
9. Dai, Z., Shrivastava, A.: Adaptive learned Bloom filter (Ada-BF): Efficient utilization of the classifier (2019). URL https://arxiv.org/abs/1910.09131
10. Ding, J., Minhas, U.F., Zhang, H., Li, Y., Wang, C., Chandramouli, B., Gehrke, J., Kossmann, D., Lomet, D.: ALEX: An updatable adaptive learned index (2019). URL https://arxiv.org/abs/1905.08898
11. Ferragina, P., Vinciguerra, G.: The PGM-index: a multicriteria, compressed and learned approach to data indexing (2019). URL https://arxiv.org/abs/1910.06169

12. Gaede, V., Günther, O.: Multidimensional access methods. ACM Computing Surveys **30**(2), 170–231 (1998)
13. Galakatos, A., Markovitch, M., Binnig, C., Fonseca, R., Kraska, T.: FITing-Tree: a data-aware index structure. In: Proceedings of the International Conference on Management of Data, SIGMOD, pp. 1189–1206. ACM, New York, NY, USA (2019)
14. Hadian, A., Heinis, T.: Considerations for handling updates in learned index structures. In: Proceedings of the Second International Workshop on Exploiting Artificial Intelligence Techniques for Data Management, aiDM, pp. 3:1–3:4. ACM, New York, NY, USA (2019)
15. Hashemi, M., Swersky, K., Smith, J.A., Ayers, G., Litz, H., Chang, J., Kozyrakis, C., Ranganathan, P.: Learning memory access patterns. In: ICML, *Proceedings of Machine Learning Research*, vol. 80, pp. 1924–1933. PMLR (2018)
16. Hsu, C.Y., Indyk, P., Katabi, D., Vakilian, A.: Learning-based frequency estimation algorithms. In: International Conference on Learning Representations, ICLR (2019)
17. Idreos, S., Zoumpatianos, K., Chatterjee, S., Qin, W., Wasay, A., Hentschel, B., Kester, M., Dayan, N., Guo, D., Kang, M., Sun, Y.: Learning data structure alchemy. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering **42**(2), 46–57 (2019)
18. Idreos, S., Zoumpatianos, K., Hentschel, B., Kester, M.S., Guo, D.: The data calculator: Data structure design and cost synthesis from first principles and learned cost models. In: Proceedings of the International Conference on Management of Data, SIGMOD, pp. 535–550. ACM, New York, NY, USA (2018)
19. Kipf, A., Kipf, T., Radke, B., Leis, V., Boncz, P.A., Kemper, A.: Learned cardinalities: Estimating correlated joins with deep learning. In: CIDR. www.cidrdb.org (2019)
20. Kraska, T., Alizadeh, M., Beutel, A., Chi, E.H., Kristo, A., Leclerc, G., Madden, S., Mao, H., Nathan, V.: SageDB: a learned database system. In: CIDR. www.cidrdb.org (2019)
21. Kraska, T., Beutel, A., Chi, E.H., Dean, J., Polyzotis, N.: The case for learned index structures. In: Proceedings of the International Conference on Management of Data, SIGMOD, pp. 489–504. ACM, New York, NY, USA (2018)
22. Li, P., Hua, Y., Zuo, P., Jia, J.: A scalable learned index scheme in storage systems (2019). URL `https://arxiv.org/abs/1905.06256`
23. Li, X., Li, J., Wang, X.: ASLM: Adaptive single layer model for learned index. In: G. Li, J. Yang, J. Gama, J. Natwichai, Y. Tong (eds.) Database Systems for Advanced Applications, pp. 80–95. Springer International Publishing, Cham (2019)
24. Macke, S., Beutel, A., Kraska, T., Sathiamoorthy, M., Cheng, D.Z., Chi, E.H.: Lifting the curse of multidimensional data with learned existence indexes. In: Workshop on ML for Systems at NeurIPS (2018)
25. Mehta, D.P., Sahni, S. (eds.): Handbook of Data Structures and Applications, 2 edn. Chapman and Hall/CRC (2018)
26. Mitzenmacher, M.: A model for learned Bloom filters and optimizing by sandwiching. In: 32nd Conference on Neural Information Processing Systems, NeurIPS (2018)
27. Navarro, G.: Compact data structures: A practical approach. Cambridge University Press, New York, NY, USA (2016)
28. O'Rourke, J.: An on-line algorithm for fitting straight lines between data ranges. Communications of the ACM **24**(9), 574–578 (1981)
29. Petrov, A.: Algorithms behind modern storage systems. Communications of the ACM **61**(8), 38–44 (2018)
30. Qu, W., Wang, X., Li, J., Li, X.: Hybrid indexes by exploring traditional B-tree and linear regression. In: Proceedings of the 16th International Conference on Web Information Systems and Applications, WISA, pp. 601–613. Springer International Publishing, Cham (2019)
31. Rae, J., Bartunov, S., Lillicrap, T.: Meta-learning neural Bloom filters. In: Proceedings of the 36th International Conference on Machine Learning, *Proceedings of Machine Learning Research*, vol. 97, pp. 5271–5280. PMLR, Long Beach, California, USA (2019)
32. Vinciguerra, G., Ferragina, P., Miccinesi, M.: Superseding traditional indexes by orchestrating learning and geometry (2019). URL `https://arxiv.org/abs/1903.00507`
33. Vitter, J.S.: External memory algorithms and data structures: Dealing with massive data. ACM Computing Surveys **33**(2), 209–271 (2001)

34. Wang, H., Fu, X., Xu, J., Lu, H.: Learned index for spatial queries. In: Proceedings of the 20th International Conference on Mobile Data Management, MDM, pp. 569–574. IEEE (2019)
35. Wu, Y., Yu, J., Tian, Y., Sidle, R., Barber, R.: Designing succinct secondary indexing mechanism by exploiting column correlations. In: Proceedings of the International Conference on Management of Data, SIGMOD, pp. 1223–1240. ACM, New York, NY, USA (2019)
36. Xiang, W., Zhang, H., Cui, R., Chu, X., Li, K., Zhou, W.: Pavo: A RNN-based learned inverted index, supervised or unsupervised? IEEE Access **7**, 293–303 (2019)
37. Zhang, M., Wang, H., Li, J., Gao, H.: Learned sketches for frequency estimation. Information Sciences **507**, 365 – 385 (2020)
38. Zhang, Y., Huang, Y.: "Learned" operating systems. SIGOPS Operating Systems Review **53**(1), 40–45 (2019)