Learned Sorted Table Search and Static Indexes in Small Model Space (Extended Abstract) *

Domenico Amato[1] Giosué Lo Bosco[1†] Raffaele Giancarlo[1]

[1]Dipartimento di Matematica e Informatica
Universitá degli Studi di Palermo, ITALY
July 7, 2022

Abstract

Machine Learning Techniques, properly combined with Data Structures, have resulted in Learned Static Indexes, innovative and powerful tools that speed-up Binary Search, with the use of additional space with respect to the table being searched into. Such space is devoted to the ML model. Although in their infancy, they are methodologically and practically important, due to the pervasiveness of Sorted Table Search procedures. In modern applications, model space is a key factor and, in fact, a major open question concerning this area is to assess to what extent one can enjoy the speed-up of Learned Indexes while using constant or nearly constant space models. We address it here by (a) introducing two new models, i.e., denoted **KO-BFS** and **SY-RMI**, respectively; (b) by systematically exploring, for the first time, the time-space trade-offs of a hierarchy of existing models, i.e., the ones in **SOSD**, together with the new ones. We document a novel and rather complex time-space trade-off picture, which is very informative for users. We experimentally show that the **KO-BFS** can speed-up Interpolation Search and Uniform Binary Search in constant space. For other versions of Binary Search, our second model, together with the bi-criteria **PGM** index, can achieve a speed-up with a model space of 0.05% more than the one taken by the table, being competitive in terms of time-space trade-off with existing proposals. The **SY-RMI** and the bi-criteria **PGM** complement each other quite

*This research is funded in part by MIUR Project of National Relevance 2017WR7SHH "Multicriteria Data Structures and Algorithms: from compressed to learned indexes, and beyond". We also acknowledge an NVIDIA Higher Education and Research Grant (donation of a Titan V GPU). Additional support to RG has been granted by INdAM – GNCS Project 2020 "Algorithms, Methods and Software Tools for Knowledge Discovery in the Context of Precision Medicine"

†corresponding author, email: giosue.lobosco@unipa.it

well across the various levels of the internal memory hierarchy. Finally, our findings are of interest to designers, since they highlight the need of further studies regarding the time-space relation in Learned Indexes.

1 Introduction

With the aim of obtaining time/space improvements in classic Data Structures, an emerging trend is to combine Machine Learning techniques with the ones proper of Data Structures. This new area goes under the name of Learned Data Structures. It was initiated by [15], it has grown very rapidly [8] and now it has been extended to include also Learned Algorithms [20], while the number of Learned Data Structures grows [4]. In particular, the theme common to those new approaches to Data Structures Design and Engineering is that a query to a data structure is either intermixed with or preceded by a query to a Classifier [6] or a Regression Model [10], those two being the learned part of the data structure. Learned Bloom Filters [15, 19] are an example of the first type, while Learned Indexes are examples of the second one [8, 15]. Those latter are also the object of this research.

1.1 Learned Searching in Sorted Sets

With reference to Figure 1, a generic paradigm for learned searching in sorted sets consists of a model, trained over the data in a sorted table. As described in Section 3.2, such a model may be as simple as a straight line or more complex, with a tree-like structure. It is used to make a prediction regarding where a query element may be in the sorted table. Then, the search is limited to the interval so identified and, for the sake of exposition, performed via Binary Search.

All of the available current contributions to this area have provided Learned Indexes, i.e., models more complex than straight lines and that occupy space in addition to the one taken by table. Such a space occupancy cannot be considered a constant, since it depends on various parameters characterizing the model.

But the use of more space to speed-up Binary Search in important Data Base tasks is not new, e.g., [22]. Consider the mapping of elements in the table to their relative position within the table. Since such a function is reminiscent the of Cumulative Ditribution Function over the universe U of elements from which the ones in the table are drawn, as pointed out by Markus et al. [17], we refer to it as CDF. Now, for the same speed-up task, the fact that one can derive a CDF from the table and approximate that curve via a regression line to make a prediction is not new, e.g., [3].

It is quite remarkable then that the novel model proposals, sustaining Learned Indexes, are quite effective at speeding up Binary Search at an unprecedented scale and be competitive with respect to even more complex indexing structures, i.e., B^+**-Tree** [5]. Indeed, a recent benchmarking study [17] (see also [12]) shows quite well how competitive those Learned Data Structures are, in

addition to providing an entire experimental environment designed to be useful for the consistent evaluation of current Learned Indexes proposals and hopefully future ones. Another, more recent, study offers an in-depth analysis of Learned Indexes and provides recommendations on when to use them as opposed to other data structures [16]. Relevant for the research presented here are the Recursive Model Index paradigm (**RMI**, for short) [15], the Radix-Spline index (**RS**, for short) [13], and the Piecewise Geometric Model index (**PGM**, for short) [9, 7]. For the convenience of the reader, they are briefly described in Section 3.2.

As already mentioned, all those models use non-constant additional space with respect to the original table. In fact, experimental studies show a time-space trade-off, which has not been investigated consistently and coherently with respect to the time-honored methodology coming from Classic Data Structures [14]. Moreover, it is missing an assessment of how good would be constant space models at speeding-up classic Sorted Table Search Procedures, i.e., Binary, Interpolation Search and their variants.

Indeed, two related fundamental questions have been overlooked. The first consists of assessing to what extend one can enjoy the speed-up of Binary Search provided by Learned Indexes with respect to the additional space one needs to use. The second consists of assessing how space-demanding should be a predictive model in order to speed-up Sorted Table Search Procedures, In particular, a constant space model would yield Learned Sorted Table Search Procedures, rather than indexes: a point of methodological importance and that has been overlooked so far. Indeed, answer to this question would put the Learned Searching in Sorted Sets Methodology at a par with respect to the classic purely Algorithmic Methodology: first, constant space algorithms and then more space demanding data structures, such as Search Trees (see [14]).

1.2 Our Contributions

To shed light on the posed related questions, we systematically analyse a hierarchy of representative models, without the pretence to be exhaustive. They range from very simple ones to State of the Art ones. We consider two scenarios. The simple one, which can be referred to as "textbook code", that uses nearly standard implementations of search methods and models. The second, much more advanced, uses the Learned Indexing methods and the highly tuned software supporting their execution, i.e. **CDFShop** [18] and Search on Sorted Data (**SOSD** for short) [17] (see also [12]). For our experimental evaluation, we generalize the one adopted in the benchmarking study, by extending the sizes of the datasets to fit in all the internal memory levels (see Section 3.4). Other than that, we adhere completely to the mentioned study.

Our findings, outlined in Sections 4-6, and in full in [2], reveal a rather complex scenario, in which it is possible to obtain speed-ups of Sorted Table Search procedures via Learned procedures that use small space, but the achievement of such a methodologically and practically important result is not so immediate from the State of the Art. Indeed, we need to introduce two new models.

- the **KO-BFS**, which is a constant space model and that can be used, in the simple scenario, to consistently speed-up Interpolation Search and Uniform Binary Search [14], this latter also referred to as branch-free Binary Search [11].
- The **SY-RMI**, which is a parametric space model that succinctly represents a set of models in the **RMI** family. In the second scenario, with as little as 0.05% additional space, it can speed-up branchy Binary Search, Eyzinger layout Binary Search [11] and be competitive in query time with respect to more space demanding Learned Indexes instances.

Finally, as a whole, our investigation systematically highlights, for the first time, the time-space trade-offs involved in the use of Learned Searching in Sorted Data, including indexes, which can be of use to users and stimulating for designers. Indeed, our findings call for further study of the time-space relation of Learned Indexes. In order to make our results replicable, we provide datasets and software used for this research in [1], in addition to the already available **CDFShop** and **SOSD**.

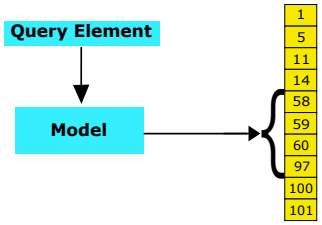

Figure 1: **A general paradigm of Learned Searching in a Sorted Set** [17]. The model is trained on the data in the table. Then, given a query element, it is used to predict the interval in the table where to search (included in brackets in the figure).

2 A Simple View of Learned Searching in Sorted Sets

Consider a sorted table A of n keys, taken from a universe U. It is well known that Sorted Table Search can be phrased as the Predecessor Search Problem: for a given query element x, return the $A[j]$ such that $A[j] \leq x < A[j+1]$. Kraska et al. [15] have proposed an approach that transforms such a problem into a learning-prediction one. With reference to Figure 1, the model learned from the data is used as a predictor of where a query element may be in the table. To fix ideas, Binary Search is then performed only on the interval returned by the model.

We now outline the basic technique that one can use to build a model for A. It relies on Linear Regression, with Mean Square Error Minimization [10]. With reference to the example in Figure 2, and assuming that one wants a linear model, i.e., $F(x) = ax + b$, Kraska et al. note that they can fit a straight line to the CDF and then use it to predict where a point x may fall in terms of rank and accounting also for approximation errors. More in general, in order to perform a query, the model is consulted and an interval in which to search for is returned. Then, to fix ideas, Binary Search on that interval is performed. Different models may use different schemes to determine the required range, as outlined in Section 3.2. The reader interested in a rigorous presentation of those ideas can consult Markus et al..

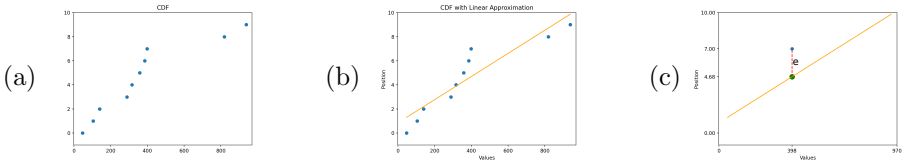

Figure 2: **The Process of Learning a Simple Model via Linear Regression.** Let A be $[47, 105, 140, 289, 316, 358, 386, 398, 819, 939]$. (a) The CDF of A. In the diagram, the abscissa indicates the value of an element in the table, while the ordinate is its rank. (b) The straight line $F(x) = ax+b$ is obtained by determining a and b via Linear Regression, with Mean Square Error Minimization. (c) The maximum error ϵ one can incur in using F is also important. In this case, it is $\epsilon = 3$, i.e., accounting for rounding, it is the maximum distance between the rank of a point in the table and its rank as predicted by F. In this case, the interval to search into, for a given query element x, is given by $[F(x) - \epsilon, F(x) + \epsilon]$.

For this research, it is important to know how much of the table is discarded once the model makes a prediction on a query element. For instance, Binary Search, after the first test, discards 50% of the table. Because of the diversity across models to determine the search interval, and in order to place all models on a par, we estimate the reduction factor of a model, i.e., the percentage of the table that is no longer considered for searching after a prediction, empirically. That is, with the use of the model and over a batch of queries, we determine the length of the interval to search into for each query. Based on it, it is immediate to compute the reduction factor for that query. Then, we take the average of those reduction factors over the entire set of queries as the reduction factor of the model for the given table.

3 Experimental Methodology

Our experimental set-up follows closely the one outlined in the already mentioned benchmarking study by Marcus et al. regarding Learned Indexes, with

some variations. Namely, in agreement with the main intent of our study, we concentrate on Sorted Table Search methods that use $O(1)$ additional space with respect to the table size. We also include the three Learned Indexes that have been extensively benchmarked in [17], in order to possibly derive versions of them that use only a fraction of additional space, granting better query times with respect to the basic Sorted Table Search procedures query times. Moreover, since an additional intent of this study is to gain deeper insights regarding the circumstances in which Learned versions of the Sorted Table Search procedure and Indexes are profitable, as a function of the main memory hierarchy and in small space, we derive our own benchmark datasets from the ones in [17].

3.1 Sorted Table Search and Classic Indexes

For this research, we use the methods and relative implementations listed and outlined below (additional technical details, as well as more literature references, are provided in [2]). The setting we consider is static, i.e., the sorted table is not modified during its lifetime.

- **Binary Search.** In addition to a standard Binary Search method, we use the best ones that come out of the work by Khuong and Morin [11] and by Shutz et al. [23]. As for terminology, we follow the one in [11]. Indeed, we refer to standard Binary Search as Branchy Binary Search (**BBS**, for short). Moreover, we refer to Uniform Binary Search [14] and its homologous routines as branch-free. Those routines diffentiate themselves from the standard one because there is no test for exit within the main loop and the remaining test is transformed into a conditional move at compile time (see [11] for details). We also include in this research, a branch-free version of Binary Search (**BFS**, for short), the branch-free Eytzinger layout (**BFE**, for short) from the study in [11], branch-free k-ary search (**K-BFS**, for short) and its branchy version (**K-BBS**, for short). We use k in $[3, 20]$, although the recommendation in that study is to use $k = 3$.
- **Interpolation Search**. As a baseline of this method, introduced by Peterson [21], we use our own textbook implementation (denoted **IBS**, for short) and **TIP** by VanSandt et al. [24]. However, we do not report results regarding this procedure due to its poor performance.
- **Classic Indexes:B-Trees** [5], in particular **B^+-Tree** [15].

3.2 Model Classes Characterizing Model Space

With the exception of the ones operating on table layouts different than sorted and the **B-Trees**, all procedures mentioned in Section 3.1 have a natural Learned version. For each, its time and space performances depend critically on the model used to predict the interval to search into. Here we consider four classes of models. The first two classes consist of models that use constant space, while

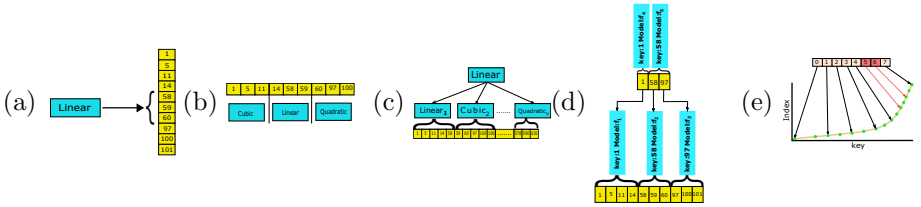

Figure 3: **Examples of various Learned Indexes** (see also [17]). (a) an Atomic Model, where the box linear means that the CDF of the entire dataset is estimated by a linear function via Regression, as exemplified in Figure 2. (b) An example of a **KO-BFS**, with $k = 3$. The top part divides the table into three segments and it is used to determine the model to pick at the second stage. Each box indicates which Atomic Model is used for prediction on the relevant portion of the table. (c) An example of an **RMI** with two layers and branching factor equal to b. The top box indicates that the lower models are selected via a linear function. As for the leaf boxes, each indicates which Atomic Model is used for prediction on the relevant portion of the table. (d) An example of a **PGM** Index. At the bottom, the table is divided into three parts. A new table is so constructed and the process is iterated. (e) An example of an **RS** Index. At the top, the buckets where elements fall, based on their three most significant digits. At the bottom, a linear spline approximating the CDF of the data, with suitably chosen spline points. Each bucket points to a spline point so that, if a query element falls in a bucket (say six), the search interval is limited by the spline points pointed to by that bucket and the one preceding it (five in our case).

the other two consist of models that use space as a function of some model parameters. For each of those models, the reduction factor is determined as described in Section 2. Moreover, as already pointed out, the **KO-BFS** and the **SY-RMI** models are new and fit quite naturally in the hierarchy that we present.

Atomic Models: One Level and no Branching Factor

- **Simple Regression**[10]. We use linear, quadratic and cubic regression models. Each can be thought of as an atomic model in the sense that it cannot be divided into "sub-models". Figure 3(a) provides an example. In particular, The corresponding learned methods are prefixed by **L, Q,** or **C**. That is, **L-BFS** denotes the branch-free version of branch-free Binary Search with a linear model to restrict the search interval.

A Two-Level Hybrid Model, with Constant Branching Factor

- **A Natural Generalization of K-BFS and K-BBS.** This model partitions the table into a fixed number of segments. For each, Atomic Models

are computed to approximate the CDF of the table elements in that segment. Finally, we assign to each segment the model that guarantees the best reduction factor. We denote such a model as **KO-BFS** or **KO-BBS**, depending on the base Binary Search routine that is being used. An example is provided in Figure 3(b). As for the prediction, we perform a sequential search for the second level segment to pick and use the corresponding model for the prediction, followed by Binary Search. The number of segments is independent of the input and bounded by a small constant, i.e., at most 20 in this study.

Two-Level RMIs with Parametric Branching Factor

- **Heuristically Optimized RMIs.** Informally, an **RMI** is a multi-level, directed graph, with Atomic Models at its nodes. When searching for a given key and starting with the first level, a prediction at each level identifies the model of the next level to use for the next prediction. This process continues until a final level model is reached. This latter is used to predict the interval to search into. As pointed out in the benchmarking study, in most applications, a generic **RMI** with two layers, a tree-like structure and a branching factor b suffices. An example is provided in Figure 3(c). It is to be noted that Atomic Models are **RMI**s. Moreover, the difference between **KO-BFS** and **RMI**s is that the first level in the former partitions the table, while that same level in the latter partitions the Universe of the elements. Following the benchmarking study, we use two-layers **RMI**s and verbatim the optimization software provided in **CDFShop** to obtain up to ten versions of the generic model, for a given table. That is, for each model, the optimization software picks an appropriate branching factor and the type of regression to use within each part of the model, those latter quantities being the parameters that control the precision of the prediction as well as its space occupancy. It is also to be remarked, as pointed out in [18], that the optimization process provides only approximations to the real optimum and it is heuristic in nature, with no theoretic approximation performance guarantees. The problem of finding an optimal model in polynomial time is open.

- **Synoptic RMI.** For a given set of tables of approximately the same size, we use **CDFShop** as above to obtain a set of models (at most 10 for each table). For the entire set of models so obtained and each model in it, we compute the ratio (branching factor)/(model space) and we take the median of those ratios as a measure of branching factor *per unit* of model space, denoted UB. Among the **RMI**s returned by **CDFShop**, we pick the relative majority winner, i.e., the one that provides the best query time, averaged over a set of simulations. When one uses such a model on tables of approximately the same size as the ones used as input to **CDFShop**, we set the branching factor to be a multiple of UB, that depends on how much space the model is expected to use relative to the

input table size. Since this model can be intuitively considered as the one that best summarizes the output of **CDFShop** in terms of query time, for the given set of tables, we refer to it as *synoptic* and denote it as **SY-RMI**.

CDF Approximation-Controlled Models

- **PGM** [9]. It is also a multi-stage model, built bottom-up and queried top down. It uses a user-defined approximation parameter ϵ, that controls the prediction error at each stage. With reference to Figure 3(d), the table is subdivided into three pieces. A prediction in each piece can be done via a linear model guaranteeing an error of ϵ. A new table is formed by selecting the minimum values in each of the three pieces. This new table is possibly again partitioned into pieces, in which a linear model can make a prediction within the given error. The process is iterated until only one linear model suffices, as in the case in the Figure. A query is processed via a series of predictions, starting at the root of the tree. Also in this case, for a given table, we have built models, i.e., ten, as prescribed in the benchmarking study and with the use of the parameters, software and methods provided there, i.e, **SOSD**. It is to be noted that the **PGM** index, in its bi-criteria version, is able to return the best query time index, within a given amount of space the model is supposed to use. We refer to this version of **PGM** as **PGM_M**.

- **RS** [13]. It is a two-stage model. It also uses a user-defined approximation parameter ϵ. With reference to Figure 3(e), a spline curve approximating the CDF of the data is built. Then, the radix table is used to identify spline points to use to refine the search interval. Also in this case, we have performed the training as described in the benchmarking study.

In what follows, for ease of reference, we refer to the models in the first two classes as constant space models, while to the ones in the remaining classes as parametric space models.

3.3 Hardware

All the experiments have been performed on a workstation equipped with an Intel Core i7-8700 3.2GHz CPU with three levels of cache memory: (a) 64kb of L1 cache; (b) 256kb of L2 cache; (c)12Mb of shared L3 cache. The total amount of system memory is 32 Gbyte of DDR4. The operating system is Ubuntu LTS 20.04.

3.4 Datasets

We use the same real datasets of the benchmarking study. In particular, we restrict attention to integers only, each represented with 64 bits unless otherwise

specified. For the convenience of the reader, a list of those datasets, with an outline of their content, is provided next.

- **amzn**: book popularity data from Amazon. Each key represents the popularity of a particular book. We have two versions of this dataset, one where each item is represented with 64 and another with 32 bits, respectively.
- **face**: randomly sampled Facebook user IDs. Each key uniquely identifies a user.
- **osm**: cell IDs from Open Street Map. Each key represents an embedded location.
- **wiki**: timestamps of edits from Wikipedia. Each key represents the time an edit was committed.

Moreover, we adapt those datasets for our research, as follows. Starting from them, we produce sorted tables of varying sizes and that preserve the CDF of the original dataset, so that each fits in a level of the internal memory hierarchy. Our choice provides a wider spectrum of experimentation with respect to the one provided in all of the Learned Indexes studies, including the benchmarking one. Given the four level memory hierarchy, each table is referred to with the suffix of that level, i.e. **amzn-L1** refers to the **L1** level cache. As for query dataset generation, for each of the tables built as described above, we extract uniformly and at random (with replacement) one million elements.

4 Learning the CDF of a Sorted Table and Mining SODS Output for the Synoptic RMI: Outline of Experiments and Findings

Models need to learn the CDF function of the table to be searched into. Regarding this point, the full set of experiments, across tables, memory levels and models, are reported in full in Section 4 of the main manuscript [2]. Due to space constraints, here we limit ourselves to report only the time required to obtain the synoptic **RMI** from the output of **CDFShop** (see Figure 4). Such a construction is performed as described in Section 3.2. The simulation to identify the relative majority **RMI**s is performed on query datasets extracted as described in the previous section, but using only 1% of the number of query elements specified there.

A full discussion of our experiments is available in Section 4 of the main manuscript [2]. Here we limit ourselves to report that the construction of the **SY-RMI** is in line with the **CDFShop** training and therefore can be profitably used as a post-processor to it. Moreover, regarding the Learning time of the **RS** and **PGM** indexes, they can be both built in one pass, which is important for Database applications [13]. According to the study just mentioned and results

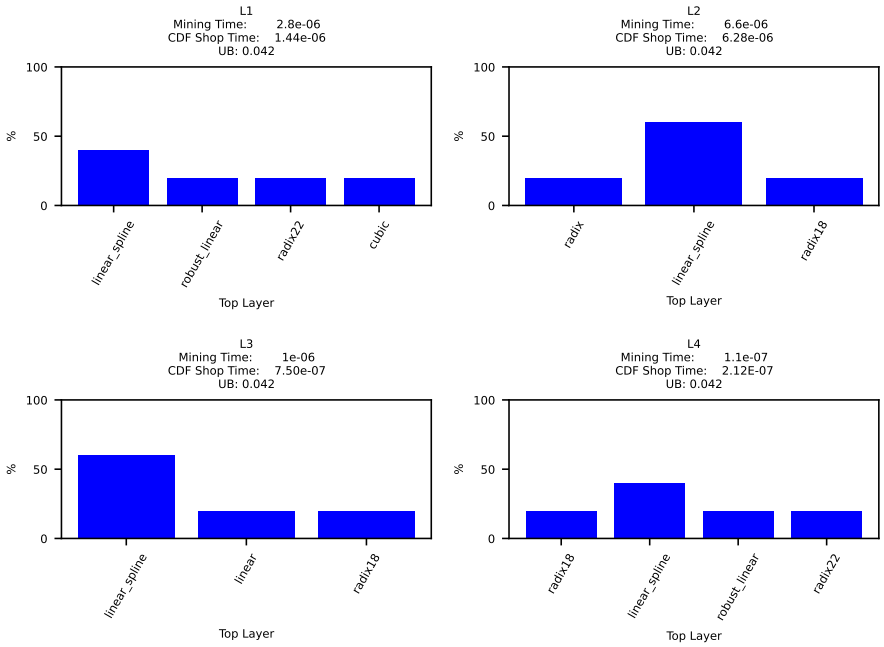

Figure 4: **Time and UB for the identification of SY-RMIs**. For each memory level, only the top layer of the various models is indicated in the abscissa, while the ordinate indicates the number of times, in percentage, the given model is the best in terms of query performance on a table. The branching factor per unit of space as well as the time it took to identify the proper **SY-RMI** (average time per element, over all **RMI**s returned by **CDFShop**) are reported on top of each figure. For comparison, we also report the same time for the output of **CDFShop.**

in [17], the **RS** is faster to build than the **PGM** index for tables fitting in main memory. Our experiments show that the **PGM** is faster to build for tables fitting each level of the cache. This is an important addition to the current State of the Art.

5 Constant Space Models: Outline of Query Experiments

This is the elementary scenario, in which we use nearly standard textbook code. In particular, the models considered in this section use constant space. The full set of experiments regarding the procedures described in Sections 3.1 and 3.2 (constant space models) have been performed on all tables considered for

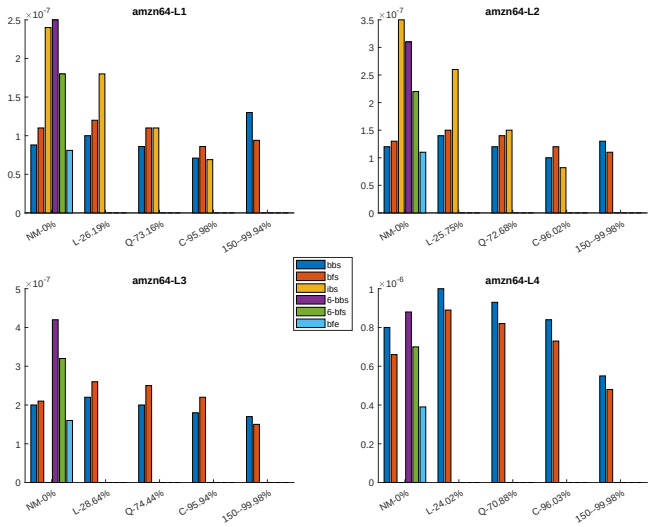

Figure 5: **Query times for the amzn64 dataset on Sorted Table Search Procedures**. The methods are the ones in the legend (middle of the four panels, the notation is as in the main text and each method has a distinct colour). For each memory level, the abscissa reports methods are grouped by model. From left to right, no model, linear, quadratic, cubic and **KO-**, with $k = 15$, and with **BFS** and **BBS** as search methods. **K-BFS** is reported with $k = 6$. For each model, the reduction factor corresponding to the table is also reported on the abscissa. On the ordinate, it is reported the average query time, in seconds. For memory level **L4**, **IBS**, **L-IBS** and **Q-IBS** have been excluded, since inclusion of their query time values (3.1e-06, 2.1e-06, 1.2e-06, respectively) would make the histograms poorly legible.

this research and reported in Section 5 of the main manuscript [2], where a detailed discussion is also present. Among all the figures documenting our query experiments, here we provide only one representative case, i.e., Figure 5.

The learned versions of Interpolation Search, together with the variants considered here, can profitably use constant space models to consistently obtain a speed-up with respect to the standard counterparts, across memory levels, and in particular with the simple models based on simple linear regression (see Section 2). As for branch-free Uniform Binary Search [11, 14] and the corresponding variant of k-ary Search [23], the speed-up can be achieved in constant space with a slightly more complex model, i.e., the **KO-BFS** introduced here (see Section 3.2). As for classic branchy Binary Search, speed-up with constant space seems to be problematic with simple regression models.

In summary, Learned Searching in a Sorted Set, with Interpolation Search or Uniform Binary Search, is fully analogous to the classic approach, with benefits in the practical performance of the former over the latter.

We also consider array layouts other than sorted, a point completely over-

looked in the research conducted in Learned Indexes. Quite surprisingly, our experiments show that none of the constant space models used in this research is able to "beat" Binary Search with an Eytzinger layout [11]. This finding has important methodological implications: it points out the need to devise Models able to speed up Binary Search with array layouts other than sorted.

6 Parametric Space Models: Outline of Query Experiments

This is the advanced scenario. In particular, the models considered here have a space occupancy that depends on parameters specific to the models. Moreover, all the experiments are supported by a highly effective software environment such as **SOSD**.

The full set of experiments is described in Section 6 of the main manuscript [2]. For the bi-criteria **PGM** and for **SY-RMI**, we have considered three space-bound: $0.05\%, 0.7\%, 2\%$. For each percentage, this is the amount of additional space the model can use with respect to the table size. As for the remaining models, including the **PGM**, we use the output of **SODS**. However, we do not consider models that use a percentage of space higher than 10% of each table size. For the remaining models, we report the one with the best query time. Moreover, we take as a baseline the **SOSD** version of **BBS**, which is implemented via vectors rather than arrays (as in the elementary case). All models use that version of **BBS** and, for consistency with the benchmarking study, we use those "branchy" models. For completeness and as a further baseline, we also include our own vector implementation of **BFS**, executed within the **SOSD** software.

The full set of results are reported in Section 6 of the main manuscript [2]. For completeness, we report in Figure 6 the same representative dataset, as for the constant space case. Query times are again averages over one million queries. Moreover, in order to gain a synoptic quantitative evaluation of the relationships among space, query time and prediction accuracy, Table 6 in the Supplementary Material of [2] (omitted here for brevity) reports the average space, query time and reduction factor computed on all experiments performed in this study, normalized with respect to the best query time model coming out of **SOSD**.

Our experiments show that both **SY-RMI** and the bi-criteria **PGM** are able to perform better than **BBS** and **BFS** across datasets and memory levels, with very little additional space.

That is, as far those two Binary Search Routines are concerned and within the **SOSD** software environment, one can enjoy the speed of Learned Indexes with very little of a space penalty.

Our study also provides additional useful insights into the relation time-space in Learned Indexes.

- **The Models Provided by SOSD with at Most** 10% **of Additional**

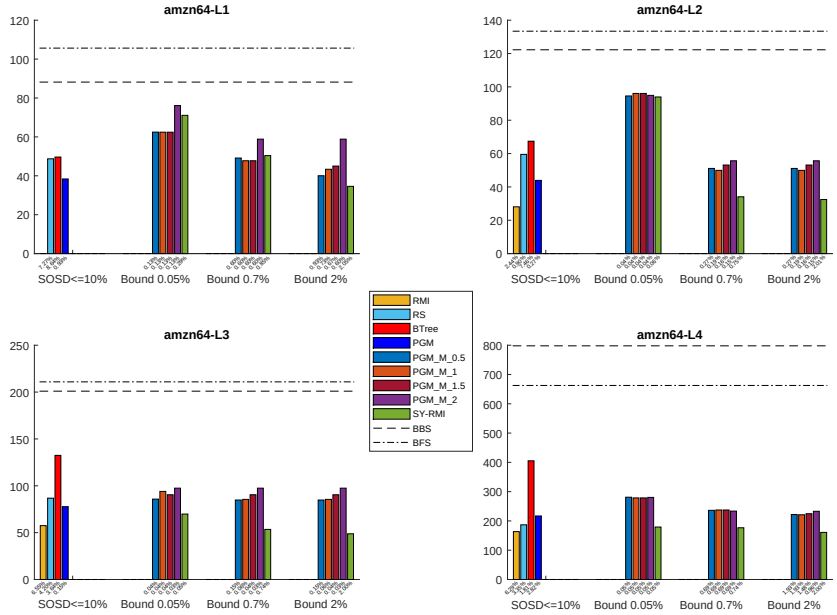

Figure 6: **Query times for the amzn64 dataset on Learned Indexes in Small Space.** The methods are the ones in the legend (middle of the four panels, the notation is as in the main text and each method has a distinct color). For each memory level, the abscissa reports methods grouped by space occupancy, as specified in the main text. When no model in a class output by **SOSD** takes at most 10% of additional space, that class is absent. The ordinate reports the average query time, with **BBS** and **BFS** executed in **SOSD** as baseline (horizontal lines).

Space. Both the **RS** and the **B-tree** are not competitive with respect to the other Learned Indexes. Those latter consistently use less space and time, across datasets and memory levels. As for the **RMI**s coming out of **SOSD**, they are not able to operate in small space at the **L1** memory level. On the other memory levels, they are competitive with respect to **PGM_M** and **SY-RMI**, but seem to require more space compared to them.

- The **PGM_M** and **SY-RMI**. Except for the **L1** memory level, it is possible to obtain models that take space very close to a user-defined bound. The **L1** memory level is an exception since the table size is really small. As for query time, the **PGM_M** performs better on the **L1** and **L4** memory levels, while the **SY-RMI** on the remaining two. This complementarity and good control of space make those two models quite useful in practice.

- **Space, Time, Accuracy of Models.** The benchmarking study provides evidence that a small model with good accuracy may not provide

the best query time. Table 6 in the Supplementary Material of [2] provides a more detailed and somewhat more articulate picture. It reports the average space, query time and reduction factor computed on all experiments performed in this study, normalized with respect to the best query time model coming out of **SOSD**. First, it can be observed that, even in small space, it is possible to obtain very good , if not nearly perfect, prediction. However, prediction power is somewhat marginal to assess performance. Indeed, across memory levels, we see a space hierarchy of model configurations. The most striking feature of this hierarchy is that the gain in query time between the best model and the others is within small constant factors, while the difference in space occupancy may be several orders of magnitude. That is, space is the key to efficiency.

7 Conclusions and Future Directions

In this research, we have provided a systematic experimental analysis regarding the ability of Learned Model Indexes to perform better than Binary and Interpolation Search in small space. Although not as simple as it seems, we show that this is indeed possible. However, our results also indicate that there is a big gap between the best performing methods and the others we have considered and that operate in small space. Indeed, the query time performance of the latter with respect to the former is bounded by small constants, while the space usage may differ even by five orders of magnitude. This brings to light the acute need to investigate the existence of "small space" models that should close the time gap mentioned earlier. Another important aspect, with potential practical impact, is to devise models that can work on layouts other than Sorted, i.e., Eytzinger. Finally, given that **BFE** within **SOSD** is consistently faster than **BBS** for datasets fitting in main memory, an investigation of **SOSD** "branchy" models (the actual ones) with respect to "branch-free" new models also deserves to be investigated.

References

[1] `https://github.com/globosco/A-learned-sorted-table-search-library`.

[2] D. Amato, G. Lo Bosco, and R. Giancarlo. Learned sorted table search and static indexes in small model space. *CoRR*, abs/2107.09480, 2021.

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

[4] A. Boffa, P. Ferragina, and G. Vinciguerra. A "learned" approach to quicken and compress rank/select dictionaries. In *Proceedings of the SIAM Symposium on Algorithm Engineering and Experiments (ALENEX)*, 2021.

[5] D. Comer. Ubiquitous B-Tree. *ACM Computing Surveys (CSUR)*, 11(2):121–137, 1979.

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification, 2nd Edition*. Wiley, 2000.

[7] P. Ferragina, F. Lillo, and G. Vinciguerra. On the performance of learned data structures. *Theoretical Computer Science*, 871:107–120, 2021.

[8] P. Ferragina and G. Vinciguerra. Learned data structures. In *Recent Trends in Learning From Data*, pages 5–41. Springer International Publishing, 2020.

[9] P. Ferragina and G. Vinciguerra. The PGM-index: a fully-dynamic compressed learned index with provable worst-case bounds. *PVLDB*, 13(8):1162–1175, 2020.

[10] D. Freedman. *Statistical Models : Theory and Practice*. Cambridge University Press, August 2005.

[11] P.V. Khuong and P. Morin. Array layouts for comparison-based searching. *J. Exp. Algorithmics*, 22:1.3:1–1.3:39, 2017.

[12] A. Kipf, R. Marcus, A. van Renen, M. Stoian, Kemper A., T. Kraska, and T. Neumann. SOSD: A benchmark for learned indexes. In *ML for Systems at NeurIPS, MLForSystems @ NeurIPS '19*, 2019.

[13] A. Kipf, R. Marcus, A. van Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann. Radixspline: A single-pass learned index. In *Proceedings of the Third International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*, aiDM '20, pages 1–5. Association for Computing Machinery, 2020.

[14] D. E. Knuth. *The Art of Computer Programming, Vol. 3 (Sorting and Searching)*, volume 3. 1973.

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

[16] M. Maltry and J. Dittrich. A critical analysis of recursive model indexes. *CoRR. To appear in: Proceedings of the VLDB Endowment*, abs/2106.16166, 2021.

[17] R. Marcus, A. Kipf, A. van Renen, M. Stoian, S. Misra, A. Kemper, T. Neumann, and T. Kraska. Benchmarking learned indexes. *Proc. VLDB Endow.*, 14(1):1–13, sep 2020.

[18] R. Marcus, E. Zhang, and T. Kraska. CDFShop: Exploring and optimizing learned index structures. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 2789–2792, 2020.

[19] M. Mitzenmacher. A model for learned bloom filters and optimizing by sandwiching. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[20] Michael Mitzenmacher and Sergei Vassilvitskii. *Algorithms with Predictions*, page 646–662. Cambridge University Press, 2021.

[21] W. W. Peterson. Addressing for random-access storage. *IBM Journal of Research and Development*, 1(2):130–146, 1957.

[22] J. Rao and K. A Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 78–89. Morgan Kaufmann Publishers Inc., 1999.

[23] L. Schulz, D. Broneske, and G. Saake. An eight-dimensional systematic evaluation of optimized search algorithms on modern processors. *Proc. VLDB Endow.*, 11:1550–1562, 2018.

[24] P. Van Sandt, Y. Chronis, and J. M. Patel. Efficiently searching in-memory sorted arrays: Revenge of the interpolation search? In *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, pages 36–53, New York, NY, USA, 2019. ACM.