IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 31 January 2025, accepted 28 February 2025, date of publication 10 March 2025, date of current version 18 March 2025.

Digital Object Identifier 10.1109/ACCESS.2025.3549626

RESEARCH ARTICLE

Fast, Robust, and Learned Distribution-Based Sorting

PAOLO FERRAGINA[1,2] **AND MATTIA ODORISIO**[2]
[1]Department of L'EMbeDS, Sant'Anna School of Advanced Studies, Pisa, Italy
[2]Department of Computer Science, University of Pisa, Pisa, Italy

Corresponding author: Mattia Odorisio (mattia.odorisio@phd.unipi.it)

This work was supported in part by the EU-H2020 "SoBigData++: European Integrated Infrastructure for Social Mining and Big Data Analytics" under Grant 871042; in part by the NextGenerationEU—National Recovery and Resilience Plan (PNRR) through the Project "SoBigData.it—Strengthening the Italian RI for Social Mining and Big Data Analytics" (M.4—C.2—I 3.1—Avviso 3264 del 28/12/2021) under Grant IR0000013 and Grant B53C22001760006; and in part by the Project "Centro Nazionale di Ricerca in High-Performance Computing, Big Data and Quantum Computing—Spoke 0: FutureHPC & BigData."

ABSTRACT Despite a long history of research and achievements in designing sorting routines, new hardware features and application requirements pose advanced challenges that computer scientists are recently tackling through novel data-aware (learned) algorithms. This paper aims to better understand the strengths and limitations of learned sorters for numerical data, by addressing two main research and engineering questions: *(Q1) Which is the best trade-off, in sorting items, between the efficacy of a "learned" model in approximating the distribution of the input data and its time-space complexity?* and *(Q2) Which algorithmic structure for distribution-based sorting is suited to leverage a learned model in order to achieve state-of-the-art performance?* To answer Q1, we implement a variant of the best-known learned model (i.e., the two-layers RMI) and, also, design a fully new learned model that turns out to be space-time efficient and efficacious in approximating adversarial input distributions. We experimentally test them over 11 datasets of 200M and 800M 64-bit floating-point items, thus offering a comprehensive answer to Q1. We then address Q2 by plugging the best-learned models from above into a distribution-based sorting scheme that leads to three new sorters whose performance is tested over 33 datasets (real and synthetic) of sizes up to 800M items. Our experimental results will show that our sorters perform better than 6 (classic and learned) best-known sorters on 29 out of those 33 datasets, thus achieving new state-of-the-art tradeoffs.

INDEX TERMS Sorting, learned sort, learned indexes, learned-based data structures.

I. INTRODUCTION

The problem of sorting items is one of the oldest and most discussed topics in Computer Science, the first algorithms are more than 70 years old [1]. Over time, the main techniques have been adapted to the evolution of the hardware and the needs of the underlying applications and users. Some of the most relevant technological changes were the introduction of the memory hierarchy, parallel processors and, more recently, SIMD architectures. Besides the evolution of the hardware, new challenges were also driven by many new use cases, like DBMS [2] and search engines [3], with the necessity to sort billions or more items in a "reasonable" amount of time. It's not surprising then, how in the past few years the topic has been dealt with by many publications in the algorithmic field that proposed novel solutions exploiting the new hardware capabilities [4], and more recently the new research and algorithm engineering frontiers opened by the game-changer advent of learned indexes [5].

The first proposal in this direction has been the Learned Sort algorithm [6], [7], a new type of distribution-based sorter that leverages a learned model of the empirical cumulative distribution function (CDF) of the input data to efficiently get an approximation of their final position in the sorted order, and then deploys it to bucket items, thus obtaining a nearly-sorted array that is finally ordered with proper algorithms, such as Insertion Sort, which is known to be effective in those situations [1]. The authors

The associate editor coordinating the review of this manuscript and approving it for publication was Ramoni Adeogun.

compared Learned Sort against well-known sorters showing that its 2.0-version (aka, Learned Sort 2.0) yields an average performance improvement over states of the art: such as IPS^{4}o, SkaSort, C++ STL sort, Radix Sort, and BlockQuicksort [7]. Other proposals came then out, which are discussed in Section II-B, paving the way for this new promising line of research and raising several interesting questions among which we select the ones that we consider the most challenging:

- (Q1) Which is the best trade-off, in sorting items, between the efficacy of a "learned" model in approximating the distribution of the input data and its time-space complexity?
- (Q2) Which algorithmic structure for distribution-based sorting is suited to leverage a learned model in order to achieve state-of-the-art performance?

In this paper, we investigate these questions by focusing on the design of *sequential* sorters for numerical values (both integers and floating points of 64 bits), and, in particular, we concentrate on distribution-based sorters (à la Learned Sort) and their bucketing problem, which we attack by designing an algorithmic scheme inspired by *interpolation sorters* (like BucketSort [8] and FlashSort [9]). Our scheme is *model agnostic* in that it can deploy different learned models depending on the data distribution and a set of delicate trade-offs among bucketing quality and inference time. We will therefore study different learned models for sorting, starting from some known ones (i.e., two-layer RMI [4] or simple linear models) and introducing some novel models that either improve the two-layer RMI adopted in Learned Sort or are fully new in their algorithmic structure by resulting space efficient, monotonic, in-place, balanced in their formed buckets, and very fast in item distribution among those buckets (denoted shortly with MLB algorithm). We describe and test all these models in Section III looking at their three main performance features (i.e., training time, inference time, and efficacy in estimating the empirical CDF of the input data) over 11 synthetic datasets drawn from different distributions [6], [7], [10] of 200M and 800M 64-bit floating-point items, so deriving hints about their ultimate efficiency and usefulness in designing a sequential sorter. This will provide a robust answer to the first main question Q1.

Given these results, the second main contribution of this paper is to design an improved version of Learned Sort that hinges on a new algorithmic structure that uses a learned model for the first-level bucketing stage (to deal effectively and sufficiently fast with the complexity of the whole input), and a set of very simple linear models for the many and variegate second-level bucketing steps (to deal very fast with smaller, and possibly more homogeneous, buckets individually). Given this structure, we select and plug the best models from Q1's answer into this new learned-based algorithmic scheme thus designing and implementing three new sorters: Learned Sort 2.1, based on our improved version of the two-layer RMI; Balanced Learned Sort, based on our new balanced linear model; and Unbalanced Learned Sort, based on simple linear interpolation models.

Then, following previous papers on this topic (such as [6], [7], [10], [11]), we will perform in Section IV a wide set of experiments that will test a total of 9 sequential sorters — i.e., 5 classic ones, including the state-of-the-art IPS^{4}o [4], and the learned sorters (known and our three new ones) — over 11 synthetic datasets (4 with duplicated keys) and 22 real datasets of sizes up to 800M keys. This plethora of experimental figures will show that our Learned Sort 2.1 is faster than Learned Sort 2.0 over all 11 synthetic datasets and over 20 out of the 22 real datasets. At the same time, it is faster than the 5 non-learned state-of-the-art sorters on all the synthetic datasets except the one with very few distinct items (where proper mechanisms that handle this kind of input could be preferable, such as in IPS^{4}o), and on 17 out of 22 real datasets (with a significant advantage on 13 of them).

As far as our two fully-new learned sorters, i.e. Unbalanced and Balanced Learned Sort, are concerned, our wide set of experiments will show that they achieve consistent and robust improvements over Learned Sort 2.1 and the non-learned state-of-the-art sorters. In particular, the Unbalanced Learned Sort improves Learned Sort 2.1 on 22 out of 33 datasets and the non-leraned IPS^{4}o on 26 out of 33. The Balanced Learned Sort improves Learned Sort 2.1 on 29 out of 33 (among the other four datasets, three of them are real datasets with 0.01% unique values or less, and the fourth one is a large real dataset with a peculiar CDF), and IPS^{4}o on 29 out of 33 (among the other four datasets, one is a large and adversarial synthetic distribution, and the other the other three are the same large real datasets with 0.01% unique values or less).

In conclusion these experimental results provide, on the one hand, a comprehensive answer to our Q2 question above and, on the other hand, they allow us to raise several new interesting research and engineering questions about the design of a highly performing sequential sorter that is robust over different input distributions, which we will discuss in Section V. The source code of all our tested sorters and benchmarking framework is publicly available on Github.[1]

II. PRELIMINARIES AND RELATED WORK

The literature on sorting algorithms is huge and, for space reasons, we cannot review it here. We refer the reader to [11] for a survey and concentrate our discussion in subsection II-B on the best-known algorithms and on the ones suitable to contextualize the design of our novel sorters. A similar consideration applies to learned indexes (see e.g. [12], [13]) that constitute the main building block of our sorters, to which we dedicate the following subsection II-A.

A. LEARNED INDEXES

Given a numerical sequence S of length n, a learned index is a data structure that approximates the *Cumulative Distribution Function* (CDF) of S in efficient time and space. These

[1] https://github.com/mattiaodorisio/balanced-learned-sort

indexes can be used to compute the *rank* of an item x in S, i.e. the number of elements in S that are $\leq x$, taking space $O(\frac{n}{B^2})$ [14] or even less [15], depending on the data distribution, rather than the classic space occupancy of $\Theta(\frac{n}{B})$ taken by B$^+$-Trees [14].

The first Learned Index, called Recursive Model Index (RMI) [5] is made of a hierarchy of *primitive models* (such as linear, polynomial etc.). At each stage, the model takes the key as input and returns the index of the model to be picked at the subsequent stage, until the final (leaf) stage predicts the rank of the key in the sorted order. The rationale is that, while going down in the hierarchy, RMI uses ''more and more specific'' models that are experts responsible for ''their'' area of the (sorted) key space to make a better prediction with a lower error. The rank prediction may be affected by an error, which is subsequently corrected using a binary/exponential search over a small area. As deeply discussed in the following, Learned Sort 2.0 [7] uses RMI made up of only two layers of linear models. After the proposal of RMI, the literature started to flourish thanks to numerous alternatives and improvements, such as ALEX [16], FITing-Tree [17], PGM-Index [18], AirIndex [19], Carmi [20], and tens of others (see [12], [13], [21], [22] and refs therein). Much is nowadays known about their main theoretical and practical aspects such as: bounding the prediction error, indexing mutable sequences, handling multi-dimensional data, up to the design of provably good *hybrid indexes* that mix classic and learned approaches [19], [20], [23].

Given their space- and time-efficient properties in effectively modeling real data distributions, the proposal in 2020 of the first learned sorter [6] based on the RMI learned model came as no surprise. Subsequently, [10], [24] observed that a desirable property for a learned model F to be used in a sorter is *monotonicity*, namely the fact that, for any pair of items $x \leq y$, it holds that $F(x) \leq F(y)$. This guarantees that the learned model preserves the *relative order* among the items to be sorted. Unless properly designed RMI is non-monotonic but still supports efficient sorting. In contrast, the simpler linear models based on interpolation are monotonic, although appearing less effective in the CDF approximation.

In the present paper, we dig into the design properties that make a learned model suitable to implement a fast sorter and eventually show that monotonicity is not the only useful property to achieve. In fact, we will identify other efficiency and efficacy properties that come into play when sorting big real datasets, so eventually devising a data-aware framework in which any learned model can be used to achieve a robust and efficient sorting performance.

B. SORTING

Among the state-of-the-art sorting algorithms, a prominent position is held by the **I**n-**P**lace **P**arallel **S**uper-**S**calar **S**ample**So**rt: IPS^{4}o [4] (2017). As the name suggests, its algorithmic structure follows the one of the SampleSort (a.k.a. multi-way quicksort, i.e., a distribution-based sorter with many pivots), and it additionally introduces a fast and in-place partitioning procedure based on hundreds or even thousands of pivots and moves items in *blocks* to exploit cache locality and parallelism.

Recently, there has been renewed interest in sorting due to the advent of learned indexes and their potential use to make the bucketing procedure ''better'' and hence faster. In particular, Kristo et al. [6] were the first to present a distribution-based sorting algorithm hinging on a *learned index* for its bucketing stage, thus named Learned Sort. In the subsequent year (2021), the same authors improved this algorithm by proposing a cache-aware partitioning procedure very similar to the one introduced in IPS^{4}o, and still driven by a learned index. They called this algorithm Learned Sort 2.0 [7].

The key idea behind Learned Sort 2.0 is to deploy the algorithmic structure of SampleSort [10] and then use a fast and space-efficient procedure to estimate (learn) the ranks of the items to be sorted in order to distribute them among a properly-chosen number of buckets. The advantage pursued with learned models, in place of a binary search among a set of pivots (possibly organized according to a Eytzinger layout [25], as implemented in IPS^{4}o [4]), resides in a fast-rank computation of the items to be distributed. Learned Sort 2.0 works in five main steps: (1) a learned index (called the *two-layer* RMI, explained in the next Section III) is trained over a sample of the input items (typically 1% of the input size); (2) all input items are distributed among a properly chosen number of buckets according to the learned-model prediction, these buckets are formed by creating first unordered blocks that are then permuted to enforce cache locality (similarly to IPS^{4}o); (3) the previous step is repeated recursively *once more* on each bucket, so to build a second level of buckets, by using the same learned model built in step 1 (this will be one of the key differences with our novel sorting scheme presented in this paper); (4) a *counting sort* based on the learned model trained in step 1 is executed on each (hopefully small) bucket of the second recursive level. Finally, (5) all second-level buckets are concatenated in proper order, and Insertion Sort is performed over the resulting *whole* input sequence to fix the items not sorted yet, possibly because of errors incurred by the (non-monotonic) learned model adopted for their distribution (as computed in step 1).

Now, despite its quadratic-time complexity in the worst case, the experiments in [7] have shown that step 5 is applied over an *almost sorted* sequence, so that Insertion Sort takes nearly (optimal) linear time thus making the algorithm very fast in practice. It is properly this efficiency and effectiveness in positioning the items in their almost-final sorted positions that raised a lot of interest in learned-based models for sorting.

After this in-memory proposal, the same authors implemented an external-memory learned sorter [24] which distributed the items in buckets sufficiently small to be sorted individually in memory, concurrently, and eventually

concatenated. This algorithmic scheme required a *monotonic* learned index, which introduced an overhead that, in any case, was affordable due to the external memory setting. These authors also addressed the sorting of ASCII records, by encoding their first 9 digits in a compact numerical representation of 8 bytes, which can be handled as an unsigned integer. If the record keys are longer than 9 bytes, this encoding scheme will not be able to capture the rest of the record numerically, and thus the sorter could be unable to fully order worst-case inputs (such as one in which all keys have the same 9-digit prefix). In 2023, Carvalho et al. [10] deeply analyzed the analogies between SampleSort and Learned Sort, and plugged a monotonic version of RMI into the ips^{4}o framework, thus designing the first *parallel* learned sorter.

We will provide a wide experimental evaluation of Learned Sort 2.0 in Section IV. We anticipate here the impact of each of the steps of Learned Sort 2.0 since this information provides useful insights into which components are worth optimizing. The model training in step (1) accounts for a non-negligible 5% of the sorting time, the first bucketing distribution (step 2) for around 30%, the overall second bucketing step (3) for 27%, the model counting sort (step 4) for 19%, and the Insertion Sort (step 5) for the 19% of the total sorting time. This timing highlights that the model training is not negligible (accounting for 5%) and the inference of the bucket where a key is routed is a costly sub-routine used many times in Steps (2), (3), and (4) which together account for 76% of the total sorting time.

Our paper therefore focuses on *sequential* learned-based sorters, concentrating on the improvement of the above (costly) training and inference steps by designing new efficient and effective learned models. In particular, we will show in Section III how to improve the existing RMI by proposing a novel training algorithm, and then we will discuss new learned models that trade the speed and accuracy of training (step 1) vs the balancedness of the bucket formation (steps 2-4). This will allow us to establish *which-when-and-how* to re-train the learned models adopted in those steps, instead of using just one single model (as done in Learned Sort 2.0). In light of these findings, we will devise three new learned-based sorters, which we call Learned Sort 2.1, Unbalanced Learned Sort and Balanced Learned Sort. As a final contribution of this paper will perform in Section IV a wide and variegated set of experiments in terms of 33 datasets (11 synthetic and 22 real) of size up to 800M items, and 9 sorters (6 known and our 3 new ones), thus eventually establishing that our novel learned-based sorting scheme and its associated models perform better than the best-known sorters on 29 out of those 33 datasets, thus achieving new state-of-the-art tradeoffs.

III. LEARNED INDEXES FOR SORTING

As stated above, the state-of-the-art learned sorter is Learned Sort 2.0 [7]. This algorithm deploys as learned index the so-called *two-layer* RMI, which consists of two layers of linear models that are trained by using a linear-spline fitting because this has been shown by the authors to guarantee better monotonicity, compared to linear regression with loss minimization. However, as stated in the previous section, this model is used as-is in both recursive levels of the distribution-based scheme adopted by Learned Sort because this reduces the training cost and still achieves well-balanced partitions, and thus efficient sorting time. In the present paper, we argue that using the same model for both recursive levels is *too restrictive* and can *potentially slowdown* the performance of the final sorter on some specific distributions and real datasets; thus, we study a set of known and newly designed learned models that offer different trade-offs in training-vs-inference speeds and buckets balancedness.

The following Section III-A recalls the main algorithmic ideas behind the two-layer RMI, how it is used in the design of Learned Sort 2.0, and then digs into our improvement of this learned model that results in faster building time and thus faster first recursive call of Learned Sort (i.e. steps 1-2 above). Because of its algorithmic features, we call this first new proposal Learned Sort 2.1.

However, we show that Learned Sort 2.1 still incurs some significant limitations that we address in Section III-B by designing a fully new, learned model whose design departs from RMI to guarantee still reasonable balancedness in the item's distribution among the individual buckets, but much faster speed in training and inference. We call this new learned model MLB, which stands for Monotonic Linear Balanced-bucketing model.

Finally in Section III-C, we compare experimentally our two new models above against some other simple linear models that offer fast training and inference, at the cost of a worse bucket balancedness. These experiments will highlight the main algorithmic and practical features of our proposals and will allow us to *cherry pick* their instantiations that will originate three new learned sorters: one called Learned Sort 2.1, based on our improved two-layer RMI, and the other two called Balanced Learned Sort and Unbalanced Learned Sort, based on a new two-level recursive scheme that uses our novel MLB model or a linear model (respectively) for the first-level bucketing (to deal effectively and sufficiently fast with the complexity and size of the whole input), and a very simple linear model for the second-level bucketing (to deal very fast with smaller, and possibly more homogeneous buckets). A win-win algorithmic scheme that will achieve improved and robust practical sorting performance, as we will show in the wide set of experiments on 11 synthetic and 22 real datasets performed in Section IV.

A. AN IMPROVED TWO-LAYER RMI

As anticipated above, the effectiveness of a learned index for sorting depends on three factors: bucketing quality, inference time, and training time. The first two are particularly critical: the algorithm needs to distribute the items evenly and quickly among the buckets under formation; on the other hand, the training time is less critical because training is done once at

the beginning of the sorting process, and over a (relatively small) sample of the input. In Learned Sort 2.0 [7], the sampling factor is fixed to 1% of the input size, we stick to this value in the rest of the paper whenever we will talk about *sampling* the input items, and defer to Section V for a discussion of its optimized setting. As said the model training should not be critical for the overall sorting performance, however, as anticipated in Section II-B, its impact in Learned Sort 2.0 accounts for 5% of the overall sorting time, thus we argue that there is room for improvements here. The obvious way to make it faster would be to reduce the number of samples, thus trading the training time with the model's accuracy. Instead, we investigated the possibility of engineering the model training without sacrificing its accuracy. Namely, we observe that a *two-layer* RMI is a C++ object containing the root model (its slope and intercept) and a vector of 1000 leaf models. At build time, it allocates a temporary buffer for each linear model. All samples are initially stored in the root model buffer and sorted using the standard library. The root linear model is trivially built by extracting the smallest and the biggest values from this buffer (the first and the last item). Then, all the items in the root buffer are classified using the root model and copied into one of the leaf buffers according to said classification. Eventually, the leaf model parameters are extracted from each leaf buffer.

Since RMI has only two layers, we provide an engineered in-place implementation of the two-layer RMI that exploits two tricks to speed up its construction and inference time: (i) the well-known swapping of all the samples at the beginning of the input; and, (ii) the fact that the linear models can be trained just connecting the minimum and the maximum items (called MinMax model), without the need to sort the training data. In particular, we use a double scan of the sample that first trains the ''root'' of the two-layer RMI (i.e., its first layer) and then distributes the sampled items among its ''leaf models'' (i.e., its second layer), while retaining the minimum and maximum of each of them. In its simplicity, this way of training the two-layer RMI introduces a 5x speed-up over the original approach of [7] (see Figure 2).

Considering that Learned Sort 2.0 adopts the same two-layer RMI in both recursive calls and our new version of this learned model can be trained much faster, we propose a new algorithmic structure for this sorter, called Learned Sort 2.1, in which we limit the use of the trained RMI model to the first bucketing procedure (steps 1-2), while we train a new simpler linear model over each bucket of the second recursive call (step 3), so to increase the bucketing quality of the original Learned Sort 2.0 in this latter secondary bucketing..[2] This is advantageous because we are training one linear model per bucket of step 3, created with the data routed from the top RMI (step 2). The approach instead taken in Learned Sort 2.0 might struggle in capturing the distribution of data within each *specific* bucket, because it uses the same RMI for all of them, and this model was trained on the *whole* sample drawn at step 1. We experimentally verify in the following Section III-C that this new approach is more convenient than Learned Sort 2.0 for all the 11 synthetic distributions (including those with duplicates), unless the case in which the input size is big (800M) in which both the algorithms fails to outperform IPS^{4}o for the *Zipf distribution*. The new approach is also more convenient than Learned Sort 2.0 over 20 out of 22 real datasets: the datasets on which this approach fails to improve the original Learned Sort 2.0 consist of 27M and 39M *duplicated reals* (with respectively only 1.18% and 0.02% unique items, few of these are highly repeated, the other are outliers), and again the non-learned competitor IPS^{4}o is anyway preferable to both.

[2]Also for the re-training, 1% of the data in the bucket are sampled uniformly at random, unless the bucket size is below a threshold (derived experimentally, i.e. 400 in our experiments). In the latter case, the linear model is trained over all the data. Please note, that this is a rare case, considering the size of our input.

B. A NOVEL BALANCED LINEAR MODEL: MLB

A learned index takes in input a key x and returns a value (typically) in the range [0, 1]. In our case, since we are interested in using a learned index for routing keys in k buckets, we multiply the returned value by k and round down the result, thus getting a bucket index in $\{0, 1, 2, \ldots, k-1\}$. We aimed at designing a *Monotonic* Learned index for items Bucketing (shortly, MLB) that will renounce slightly to bucket quality (balancedness), but it will guarantee monotonicity at the bucket level, and a significantly faster construction and inference time than RMI's. Then, we plug MLB into the algorithmic structure of Learned Sort 2.1 in place of RMI to trade bucket quality with training/inference speed. We call this newly learned sorter Balanced Learned Sort (shortly, BLS).

Let us now dig into the key algorithmic ideas underlying MLB's design. First, it trains a linear MinMax model over a sample of size s of the input keys (we take $s = n/100$ as in Learned Sort), thus deriving its minimum and maximum values, and in turn its slope α and intercept β. This linear model is defined over an *expanded* co-domain of size mk, instead of k (i.e., the number of buckets), for some overpartitioning factor m to be studied and defined at experimental time (see Section IV). The value of m will depend on the data distribution and the architectural features of the machine on which the sorter will be executed. The former can be approximated somewhat by sampling, whereas the latter can be estimated via a small set of benchmarks to be executed off-line in advance.

The idea of *overpartitioning* is indeed not new, in [26] it has been successfully used in the parallel setting: r groups of *Processing Elements* (PEs) create kr buckets that are subsequently aggregated so to assign k buckets to PE-groups in a load-balanced way. Please note that [26] addresses the problem of balancing the already created buckets among PEs, here we are devising a classifier that produces balanced buckets.

The linear MinMax model is used to partition the space of the input keys into mk bins, of possibly different sizes,

evaluated via an array of counters $C = [c_1, c_2, \ldots, c_{mk}]$ such that c_i counts how many samples fall in the i-th partition: i.e., how many samples x are such that $i = \lfloor \alpha x + \beta \rfloor$ (we use the term *partition* to mark the difference from the term *bucket* used by the sorting procedure). We postulate that, since C is computed on the samples, each c_i is a good estimate of the bucket size, provided that the over-sampling factor m is chosen in a principled way [27]. MLB then splits C into k sub-arrays, of possibly different lengths, such that the sum of their counters (and thus the size of the k induced buckets) is as balanced as possible.

This splitting process boils down to a well-known optimization problem that we solve, as in [26], through a simple and optimal algorithm (see [26] for the proof) that performs a binary search to determine a value $v \in [s]$ such that we have close to k sub-arrays whose sum of their counters is bounded above by v (the case of single elements larger than v induces singleton sub-arrays). We notice that, in the case of dense buckets or special distributions, less than k non-empty sub-arrays might be formed. The binary search quickly discards extreme cases inducing highly unbalanced partitions. Its time complexity is $O(mk \log s)$, where m and k do not depend on the input size, while $s = \frac{n}{100}$. Not surprisingly, the execution of this algorithm takes a negligible time compared to the whole time for sorting and achieves well balanced buckets.

The final step of MLB is to derive a table T that maps a partition-ID from the oversized space $\{1, \ldots, mk\}$ to a bucket-ID in $\{0, 1, \ldots, k-1\}$. This way, the value $T[\lfloor \alpha x + \beta \rfloor]$ is the bucket-ID for item x, where α and β are the slope and intercept of the linear MinMax model. The monotonicity of MLB guarantees that items in a bucket are smaller than, or equal to, items in subsequent buckets. This ensures that the sorted sequence can be produced by limiting the final sorting step to the individual buckets; nevertheless, as it will be clear next, for efficiency reasons (in practice) we will run an Insertion Sort over the whole sequence produced by concatenating all buckets, this is guaranteed to move (cache-efficiently) items in small array windows.

Figure 1 shows the overall final structure of MLB which is made by the linear MinMax model and the *table* T, thus taking $mk + 2$ memory cells. In the figure, the number of buckets k is 5 and the overpartitioning factor m is 2, so the table T has size 10. The binary search finds $v = 8$ because that induces exactly $k = 5$ buckets. As stated, the partitions are not perfectly balanced, as their size are respectively: 5, 7, 10, 8, 8. The red arrows in the Figure show the step executed to compute the bucket of a key that is predicted by the MLB model to fall in the bucket $T[2] = 1$.

As mentioned in Section II-B, our Learned Sort deploys a learned model to build balanced buckets, which in our case is either the improved RMI or MLB. The distribution step is executed for a fixed number of iterations, and eventually deploys Insertion Sort applied on all concatenated buckets. Thus the correctness of the algorithm simply derives from the one of Insertion Sort. In addition, the following observation holds: if we are not in the pathological case of all equal

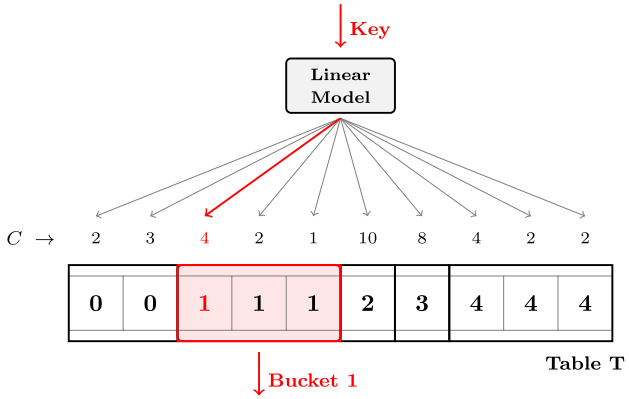


FIGURE 1. LBM architecture with $k = 5$, $m = 2$, built over a sample of size $s = 38$. The red arrows show the steps to predict the bucket of a given key, which turns out to be $T[2] = 1$). The partitioning of array C is done by the value $v = 8$.

samples, MLB is trained, and Learned Sort produces at least two non-empty buckets. The simple proof is as follows: if there exist at least two distinct samples, the minimum and maximum are distinct, hence the training of the root linear model guarantees that $c_1 > 0$ and $c_{mk} > 0$, and hence the balance procedure finds $k' \geq 2$ buckets which will be non-empty. This guarantees that the recursion can continue on k' partitions. Otherwise, in the pathological case of all equal samples, the algorithm resorts to `std::sort` from the standard library. The following Section IV will prove the practical efficiency of this algorithmic scheme.

C. MODEL EVALUATION

In this section, we compare the efficiency (training and inference time) and efficacy (bucket balancedness) of the three learned models described above: the original two-layer RMI [7], our improved two-layer RMI (Section III-A), and the fully new MLB (Section III-B). For completeness, we also test the simple linear MinMax learned model. The goal is to derive their algorithmic features that will help us in setting their parameters (i.e., bucket number k and overpartitioning factor m) and, eventually, explain the experimental results obtained in the following Section IV for the (corresponding) learned sorters: Learned Sort 2.0 (using the original two-layer RMI), our Learned Sort 2.1 (based on our improved two-layer RMI), the new Balanced Learned Sort (based on our new MLB), and the new Unbalanced Learned Sort (based on the linear MinMax model applied to both bucketing levels).

D. MODEL PERFORMANCE

We tested the four learned models over several synthetic datasets of up to 800 million keys drawn according to different distributions (the same used later in Section IV), finding of particular interest the case of unbalanced ones, being a sort of *adversarial* case for a learned model aimed at building balanced buckets. We present in detail the case of the Exponentialdistribution ($\lambda = 2$), because the others yield a coherent experimental behavior, and the observations presented here are valid there too. For completeness, we also

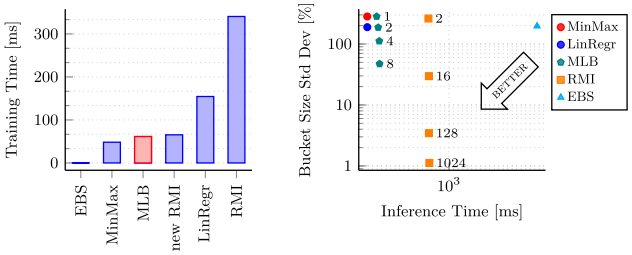


FIGURE 2. Performance comparison among 6 learned models: training time (left) and inference time/quality of their bucketing process (right) over an input of 800 million of exponentially distributed doubles. The y-axis of the right plot reports the ratio between the standard deviation of the bucket size to its optimal value (i.e., n/k); while the integer labels on the right of each icon denote the overpartitioning factor m for MLB, or the number of leaves for the two-layer RMI (here shortened as RMI).

tested two other known models, namely a simple linear regression model (shortly LinRegr); and the non-learned approach based on the *Eytzinger Binary Search* (ebs), adopted in IPS^{4}o. This way, we have a pretty complete picture of six distribution schemes driven by learned and non-learned approaches.

We recall that our study concerns *sequential* sorters, so we drop the P from IPS^{4}o and shortly denote its sequential version with IS^{4}o.

Figure 2 reports on the left the time to train the model (in milliseconds) and, on the right, the relation between the time to compute the bucket index of 800M input keys among 1024 buckets and the standard deviation of the bucket size (expressed in percentage as the ratio with its average value, i.e. n/k). Please note that the right figure depicts the bucketing efficiency and effectiveness of every model, namely one recursive level of each learned sorter, by just counting how many items are placed in each bucket and the speed to route them, but not the time needed to move them.

The Eytzinger layout is very fast to build because it is in-place and thus it needs to store just the sampled items.[3] However, although its layout is cache-efficient by design in performing the binary search, the inference still requires traversing this (array) structure for every item to distribute, thus resulting very slow and obtaining highly variable bucket sizes.

As far as the learned models are concerned, the MinMax model is the fastest among them because of its simplicity, both in training (it just needs to interpolate between the minimum and maximum values of the 1%-sized sample), and in inference (it computes just one linear equation), but it also experiences high variability in the size of the buckets.

The linear regression model (LinRegr) has highly variable buckets' size and significantly higher training time, mainly due to the need to sort the sample (we used IS^{4}o), and its inference time is the same as MinMax.

For what concerns the original two-layer RMI model and our improved version (denoted in Figure 2 as *new* RMI), we notice that ours is 5x faster in training, and, as expected, both take the same performance at inference time (this is why only one RMI is depicted on the right plot of Figure 2). In addition, both RMI's versions can obtain bucket sizes that are close to average even with a small number of leaves (e.g., 128).

On the other hand, our new MLB model is slightly faster in training and much faster in inference than both versions of the two-layer RMI, but it experiences a larger bucket size standard deviation. However, the first two positive features will largely counter-balance this latter issue thus not penalizing the overall sorting time of our novel Balanced Learned Sort, as we will show experimentally in the next Section IV. The reason is that, when sorting, the inference is repeatedly computed for each item, serving as a fundamental sub-routine of steps (2), (3), and (4), which together account for 76% of the total sorting time of the classic Learned Sort 2.0 as discussed in Section II-B.

E. SPACE OCCUPANCY

The amount of space occupied by a learned model is the other key feature to be optimized, because it may impact (un)favorably on the inference time: the smaller it is, the more likely to fit the model in the L1 cache, and the larger the remaining space available to handle the buckets during the distribution process.

In detail, the space used by the experimented (learned and non-learned) models are the following:

- IS^{4}o: a copy of the pivot items stored according to the Eytzinger layout.
- Linear MinMax and LinRegr: two real values needed to store the line slope and its intercept;
- RMI (both *new* and the original one): two real values for the line stored in the root model, plus two real values for the line stored in each leaf model;
- MLB: two real values for the line stored in its root MinMax model, plus $m\,k$ integer values in the range $[0, k-1]$ for the table T;

To set the parameters k and m in our MLB model, we observe that L2 caches can nowadays easily support up to $k = 1000$ buckets (which is the default value selected by [7] for Learned Sort 2.0). Moreover, we found experimentally that, due to memory alignment, we achieve faster access by storing 4-byte integers instead of the 2-byte integers needed to fit the $\log k$ bits of a bucked-ID. Therefore, the resulting MLB table size is $4m$ KB. Finally, we experimentally found that the best performance for an unknown distribution and on our architecture[4] is achieved when the overpartitioning factor is $m = 4$, so the overall MLB space occupancy for

[3]We used the default IS^{4}o configuration [4], that for our input size produces 256 buckets, picking the pivots with an overpartitioning factor equal to 5. This amount of samples is still negligible compared to the 1% sample size processed by the other models. It is important to observe that we considered the plain sequential Eytzinger binary search, as implemented in the IS^{4}o open-source library to be fair with the other distribution learned-based methods.

[4]Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz, 512GB ram, 32KB L1, 1024KB L2, 39MB shared L3.

$k \leq 1000$ buckets is up to 16 KB, which is the same space used by the default configuration of RMI with 1000 leaves and double precision floating points values.

In conclusion, these experiments show that: *There is an interesting trade-off offered by the tested learned models between their inference time and bucketing balancedness. On the Pareto front of Figure 2 (right) we find our novel* MLB *and both versions of the two-layer* RMI. *It is interesting to note that* MLB *is the fastest in training and inference, but gets slightly larger buckets. Conversely, the two-layer* RMI *builds more balanced buckets but it takes slightly longer training and much larger inference time. Both models are space-efficient and their data structures can be easily fit in an L1 cache.*

The following section will show that the two nice features of our new MLB model will impact favorably on the final winning performance of our Balanced Learned Sort, which will turn out to be a robust and significant performing learned sorter.

IV. EXPERIMENTAL RESULTS

We start by studying and tuning the performance of the newly introduced sorters — namely Learned Sort 2.1, Balanced Learned Sort, and Unbalanced Learned Sort — over 7 synthetic distributions. We will then move to execute a wide set of experiments over the tuned learned sorters and 5 classic non-learned sorters (including the state-of-the-art IPS^{4}o [4]) over 11 synthetic datasets (4 with duplicated keys) and 22 real datasets of sizes up to 800M keys.

We notice that the bucketing implemented by Unbalanced Learned Sort is the same as the one implemented by Balanced Learned Sort if we had set the overpartitioning factor as $m = 1$ (i.e., no overpartitioning). This will allow us to investigate whether and how much the *rebalancing step* introduced with the overpartitioning factor m and the table T of Section III-B is worth and in which situations. As a notice, we expect MinMax (used by Unbalanced Learned Sort) to be less effective in modeling the skewed CDF although being very much space efficient; on the contrary, RMI and MLB should be more efficacious at the cost of occupying a larger amount of space in memory. More precisely, the space used by MLB depends on the number k of buckets, investigated below, whereas the number of RMI leaves is set to 1000 as in Learned Sort 2.0. Finally, we observe that by comparing these three implementations of learned sorters, we are indeed experimenting with the best learned models on the Pareto front of Figure 2.

We compared the performance of these three sorters by varying the number k of buckets over 7 synthetic datasets formed by 200M and 800M 64-bit doubles drawn according to the following distributions: Chi-Squared ($k = 4$), Exponential ($\lambda = 2$), Lognormal ($\mu = 0, \rho = 0.5$), Normal ($\mu = 0, \rho = 1$), Uniform[0, 1), Gaussian mixture (a random additive distribution of five Gaussians, whose weight for the sum, means, and variances are chosen at random), Zipfian (a distribution that generates a number $x \in [1, 10^2]$ with a probability proportional to $\frac{1}{x^{0.75}}$, it models real-world data such as word frequencies in a language or the popularity of web pages [28]). Each experiment consists of 25 repetitions, then taking the average time (in seconds) for every algorithm. The benchmarks are compiled with gcc v11.4 with optimization flags -O3 and -march=native enabled. They are executed (pinning the thread on the core) on a machine equipped with Intel(R) Xeon(R) Gold 6238R CPU @ 2.20GHz, with 512GB of memory, 32KB L1, 1024KB L2, 39MB shared L3. Turboboost was disabled to reduce variability in CPU frequency.[5] This choice was driven by the need to appreciate small differences among the execution times of the tested learned-based algorithms. However, by doing this, we found that Radix Sort and SkaSort could obtain on some datasets a slight advantage wrt to IS^{4}o. Although this could be seen as not pretty consistent with what is known in the literature about IS^{4}o and classic sorters, we preferred to stick to this approach for benchmarking since it makes our tests repeatable, it allows us to investigate deeper the performance of learned sorters, and it does not anyway jeopardize our overall conclusions on the best sequential sorters.

For space reasons, we report in Figures 3 and 4 just four (out of seven) of those datasets for input size respectively of 200M and 800M, the plots for the other three datasets are reported in Figures 9 and 10 in the Appendix (and they are anyway "considered implicitly" in the following figures). The plots of Figure 3 show that Balanced Learned Sort (red line, BLS) gets faster performance independently on the number of buckets over the extremely unbalanced distribution (subfig. (d)), and with a lower number of buckets on the other distributions. The difference in time performance between Balanced Learned Sort and Unbalanced Learned Sort is very small overall (often below 100ms for 200M input), unless for very skewed distributions. However, increasing the input size as done in Figure 4 Balanced Learned Sort gets faster performance with a lower number of buckets uniformly over all the datasets. Conversely, although the two-layer RMI offers more balanced buckets (blue line, LS2.1), its lower speed in training and inference (shown in Figure 2) impacts negatively on the overall speed of Learned Sort 2.1. Looking at Unbalanced Learned Sort (green line, ULS), we notice that it offers a variegated behavior that depends on the too-simple linear model adopted at the top level of its bucketing strategy. It is also important to notice the similar behavior between BLS and ULS in the case of a dataset that is both big and extremely unbalanced (Subfigure (d) of Figure 4). As pointed out in the next section, this is the case in which the fixed structure of learned sorters has its main limit, and all the three algorithms depicted are indeed outperformed by the non-learned IS^{4}o.

[5]The source code of all our tested sorters and benchmarking framework is publicly available on Github: https://github.com/mattiaodorisio/balanced-learned-sort.

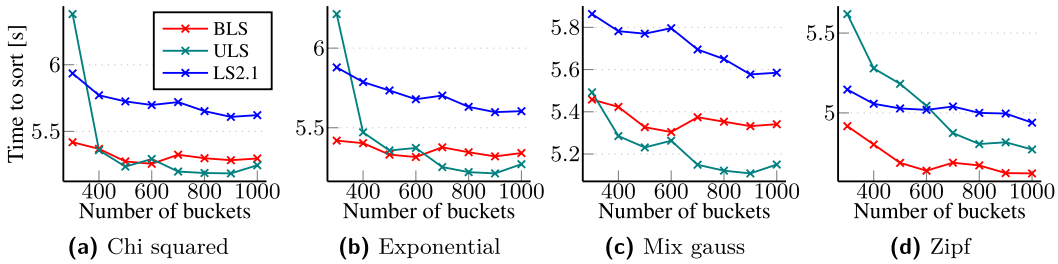


FIGURE 3. Time to sort 200 million items of the three learned-based sorters: Learned Sort 2.1, Unbalanced Learned Sort and Balanced Learned Sort, described in the text, by varying the number of buckets in the first-level recursion.

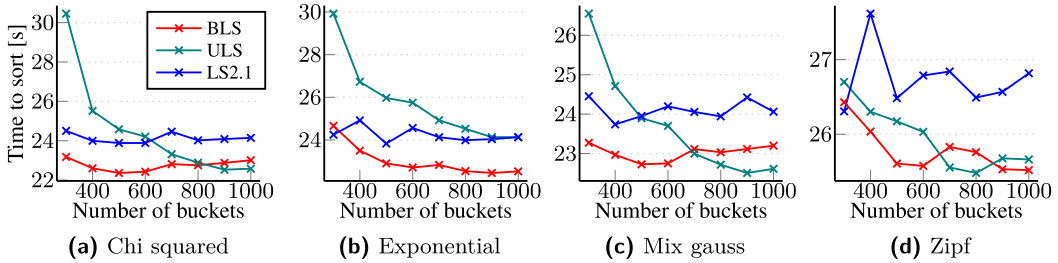


FIGURE 4. Time to sort 800 million items of the three learned-based sorters: Learned Sort 2.1, Unbalanced Learned Sort and Balanced Learned Sort, described in the text, by varying the number of buckets in the first-level recursion.

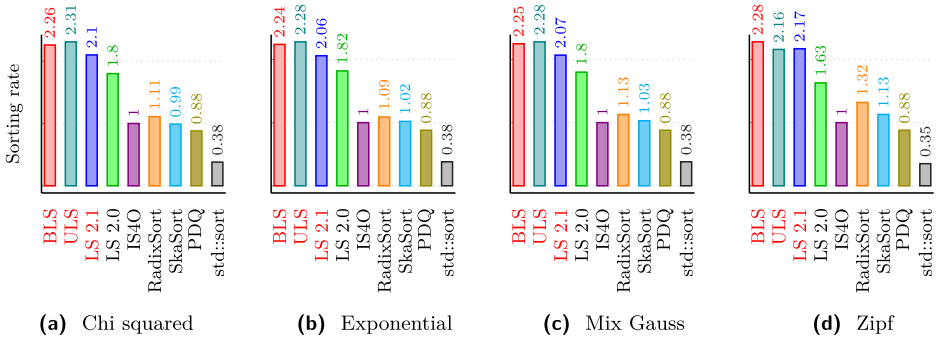
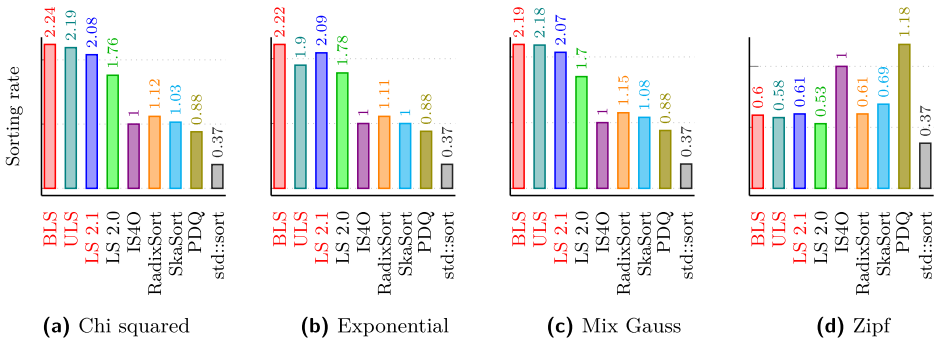


FIGURE 5. Sorting rate (the higher the better) on 200 million of doubles drawn according to four synthetic distributions, expressed as the ratio with respect to the performance of IS^{4}o. The results on the other synthetic datasets are reported in Figure 11 of Appendix A. In red are the names of the learned sorters proposed in this paper.

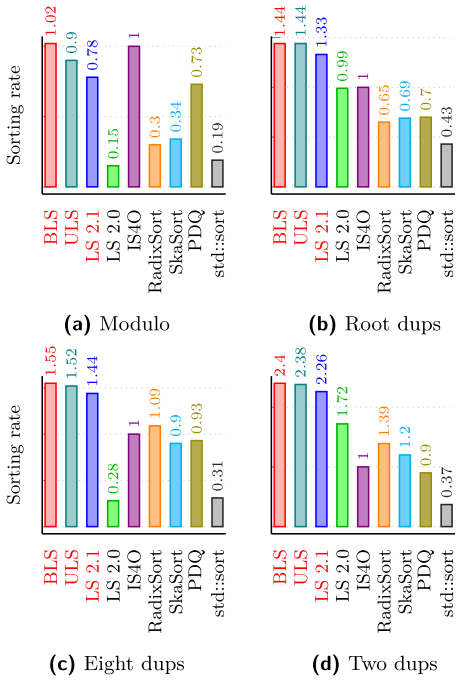
Therefore, our algorithmic choice to *balance* the result of a linear MinMax model seems to be more robust as the lowest red line in the plots highlights; however a deeper investigation of the number k of buckets is needed to better appreciate the improvement of balanced wrt unbalanced buckets. In fact, each model has its optimal value of k: for example, a linear model finds greater benefit with a higher k than MLB; but, on the other hand, k defines the number of blocks that should be kept in cache to avoid a potential cache miss for each item during the data distribution, thus k is very architecture-related. The best k also varies depending on the distribution, which is unknown in advance. By looking at the experimental results in Figures 3, 4 and 9, 10 in Appendix A, we selected a single value for each algorithm (and defer to Section V for a discussion about this issue): $k = 600$ for Balanced Learned Sort, $k = 1000$ for Unbalanced Learned Sort, and $k = 500$ for Learned Sort 2.1.

We are now ready to benchmark four learned sorters against five other (non-learned) sorters for a total of 9 sorting algorithms over 11 synthetic (Subsections IV-A – IV-B) and 22 real-world datasets (Subsection IV-C) of various sizes up to 800 million items. The learned sorters include the two versions of Learned Sort (namely the known Learned Sort 2.0, and our improved version Learned Sort 2.1), and the two variants of our novel learned sorter based on the balanced MLB model (i.e., Balanced Learned Sort) and the variant that does not *balance* the buckets (i.e., Unbalanced Learned Sort, shortly ULS). The non-learned sorters are the ones tested in [6] and [11], they are interesting either for their algorithmic structure or for their very competitive performance on some datasets. They include: IS^{4}o, which is the state-of-the-art sorter to date (here in its sequential version [4]); the Pattern Defeating Quicksort (PDQ) [29], which is a two-way Quicksort (deterministically) avoiding the worst



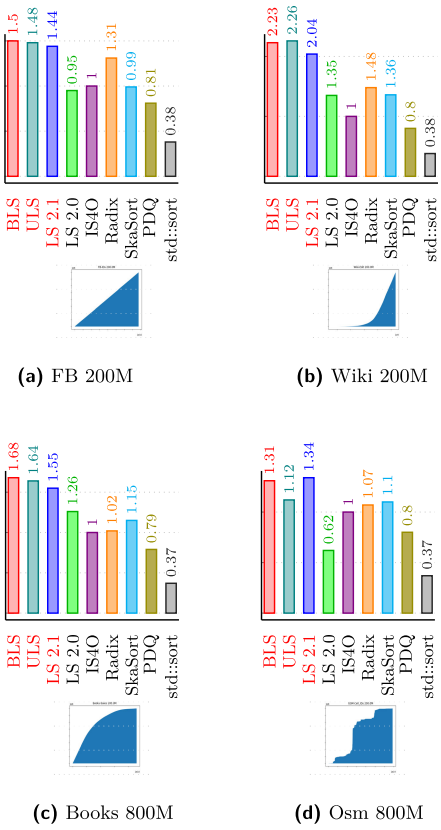
(a) Chi squared (b) Exponential (c) Mix Gauss (d) Zipf

FIGURE 6. Sorting rate (the higher the better) on 800 million of doubles drawn according to four synthetic distributions, expressed as the ratio with respect to the performance of IS^{4}o. The results on the other synthetic datasets are reported in Figure 12 of Appendix A. In red are the names of the learned sorters proposed in this paper.



(a) Modulo (b) Root dups (c) Eight dups (d) Two dups

FIGURE 7. Sorting rate (the higher the better) on 800M of doubles drawn according to four synthetic distributions with duplicates, expressed as the ratio wrt the performance of IS^{4}o.Number of unique items: 16 in Modulo, ~14k in Root Dups, ~4.21M in Eight Dups, ~22.4M in Two Dups. In red are the names of the learned sorters proposed in this paper.



(a) FB 200M (b) Wiki 200M (c) Books 800M (d) Osm 800M

FIGURE 8. Sorting rate (the higher the better) on real datasets with 200M and 800M items, expressed as the ratio with respect to the performance of IS^{4}o. The dataset CDF is depicted below each histogram. In red are the names of the learned sorters proposed in this paper. The results of the other 12 real datasets are reported in Appendix B.

case, branch mispredictions, and falling to other sorters when convenient; the SkaSort [30], which is a well-engineered most-significant-digit-first Radix Sort, interesting because learned sorters may be seen as distribution-aware radix-based sorters with a learned model behind; the classic Radix Sort [31], based on the least-significant-digit-first approach; and, finally, std::sort [32], which is the standard library from c++, not interesting for performance, but widely known.

A. BENCHMARK ON SYNTHETIC DATA

Figure 5 (and the rest of the experiments on synthetic datasets reported in Figure 11 of Appendix A) shows that the retraining introduced by our Learned Sort 2.1 gets a speed-up wrt the original Learned Sort 2.0 that goes from 13% to 33%. Surprisingly, by substituting RMI with the simpler and faster MinMax model in the first recursive level, Unbalanced Learned Sort (ULS) gets even faster performance than Learned Sort 2.0 (LS2.0), unless for the extremely unbalanced Zipf distribution, and moreover it results from 2.16 to 2.34 times faster than IS^{4}o. By introducing the novel MLB model in the first recursive level, Balanced Learned Sort (BLS) obtains the best performance on the unbalanced Zipf distribution (coherently with the previously discussed

result in Figure 3) while introducing some slight overhead on all the other synthetic distributions. Significantly enough, increasing the input size to 800M, as in Figure 6 (and Figure 12 of Appendix A), BLS obtains the best performance uniformly over all the synthetic datasets except the Uniform, Normal and Zipf distributions. In the Uniform and Normal distributions, BLS is the second fastest one, just close to ULS because of the *rebalancing*, which is not needed but still tried. In the Zipf distribution, it is interesting to note that all the learned sorters fail compared to IS^{4}o because of their fixed structure with two recursive steps. In order to deal with this case, learned sorters could introduce simple checks on the size of the buckets and further recurse before Insertion sort. In conclusion, apart from this last dataset, our newly proposed learned-based sorters show consistently and robustly their superiority with respect to the other "classic" sorters. In the next Section IV-C, we will show that the robustness of BLS will prove important for achieving consistently fast performance on real datasets.

B. HANDLING DUPLICATES

We also experimented with the 9 sorters on 4 synthetic distributions with many duplicates, as done in [6], [7], [10], and [11]. They are called: Modulo ($k = 16$), Two Dups ($A[i] = i^8 + N/2 \mod N$), Eight Dups ($A[i] = i^2 + N/2 \mod N$) and Root Dups ($A[i] = I \mod \sqrt{N}$). Figure 7 reports the experiments on 800 million duplicated doubles for all these distributions.

The code released of Learned Sort 2.0 struggles with "Eight Dups" and "Modulo" because the sampling takes items by jumping with a fixed offset (100 items to get 1%), resulting in poor representativeness on very repetitive inputs. In these cases, the number of unique items is insufficient to train RMI adequately, thus the algorithm resorts to calling the poor-performing `std::sort`.

As far as our learned-based sorters are concerned we notice that they adopt a random sampling strategy that typically gets a more representative training set, and experimental figures show in addition that MLB turns out to be very robust even with duplicates.

In conclusion, if the number of unique items is very low (as in "Modulo"), IS^{4}o is the fastest and most robust because it implements proper mechanisms to manage duplicates; but in the other three datasets, our two (balanced and unbalanced) learned sorters are the fastest, with the new *balancing stage* that pays less here because of the presence of duplicates.

C. BENCHMARK ON REAL DATA

We experimented with the 9 sorters also over a total of 8 types of real datasets of various sizes, ranging from a few tens of millions to 800M items, for a total of 22 tests. For space reasons, Figure 8 reports the tests over 4 datasets of 200M and 800M items, whose CDF is depicted below the plot. The other tests and datasets are described in Figures 13–19 of Appendix B.

Our three learned sorters are still the fastest over 20 out of 22 real datasets, with a speedup over IS^{4}o which may be up to 2.43 (NYC-Pickup). They are slower than IS^{4}o on just two real datasets: NYC-Tot and Chic-Tot, which are however relatively small (26.6M and 39M items, resp.), with a few thousand distinct items, distributed in a skewed way. Among the distributions depicted in Figure 8, it is worth observing that the CDF of OSM, in subfigure (D), is very peculiar, thus the Unbalanced Learned Sort fails in capturing it. The faster algorithm here is Learned Sort 2.1 which uses the more expressive model. On the other hand, Balanced Learned Sort achieves almost the same performance as Learned Sort 2.1, while its simplicity pays also on the other datasets in Figure 8.

In conclusion, we can state that in light of this wide and variegated set of experiments, our novel algorithmic approach to learned-based sorting (described in Section III) is robust and achieves uniformly overall synthetic and real datasets either the best or the second best performance. In particular, on all the 22 real datasets, Balanced Learned Sort is the best on 14, Unbalanced Learned Sort on 5, Learned Sort 2.1 on 1, and IS^{4}o on 2. When Balanced Learned Sort is not the fastest sorter, it is the second one over 20 out of 22 with a slowdown factor of at most 1.15. The remaining two datasets where Balanced Learned Sort is not among the top-2 are NYC-Tot and Chic-Tot, but on them, any learned sorter is not competitive against IS^{4}o.

As a final note, although Unbalanced Learned Sort is a fast algorithm on several real datasets, its performance is not robust (e.g. the datasets OSM, NYC-Dist, and Stks-Date) and, indeed, its simplicity and speed are not sufficient to guarantee acceptable performance. In contrast, MLB is overall much more reliable and this justifies its more principled design.

V. CONCLUSION AND FUTURE WORKS

In this paper, we investigated the following main question: Which algorithmic framework for distribution-based sorting is suited to leverage a learned model in order to achieve efficient performance? Our experimental results over 33 datasets (11 synthetic and 22 real), whose size is up to 800M items, showed that our learned-based sorters are the best performers and, that, the Balanced Learned Sort is the most robust sorter among them. Yet, we foresee conducting an extensive analysis to determine the best configuration of those sorters, possibly on an AI-based approach (see below): such as finding a more systematic method for choosing the number k of buckets, the overpartitioning factor m, and the sample size s, based on both the data characteristics and the available hardware resources.

It goes without saying that several research and algorithm-engineering questions remain to be answered especially in the light of the results we presented on real distributions.

The first and most prominent question left open, in our opinion, regards the design of a hybrid, robust, and highly

performing sequential sorter. We foresee an application of *learned-based tools* not only to implement the distribution stage of the sorting process, as investigated in the present paper and the literature to date, but also *to choose on-the-fly* the learned models to be used in the two recursive levels of that distribution stage (among e.g. MLB, MinMax, or RMI). There, one could deploy *features* like the input distribution, the number of allowed buckets, and the architecture of the underlying machine, etc. etc. This study could also provide wider experimentation over a larger set of inputs, including adversarial data, highlighting even more the strengths and limitations of each of the learned and non-learned sorters presented here.

Another interesting question to be addressed regards the simplicity of our learned-based algorithmic scheme that makes it suited to be extended to the parallel case, by possibly following the existing IPS^{4}o framework [4], [10], and to variable-length strings, by possibly following a radix-based framework, this way surpassing the limitations of currently simple ASCII-to-int encodings [24].

APPENDIX A
ALL SYNTHETIC DATASETS: EXPERIMENTAL RESULTS

We experimented with nine sequential sorters, learned-based and classic, over a total of 11 synthetic datasets of various sizes of 200M and 800M items 64-bit doubles: 7 including distinct items and 4 including (from a few to many) duplicated items. The former 7 datasets are drawn according to the following distributions: Chi-Squared ($k = 4$), Exponential ($\lambda = 2$), Lognormal ($\mu = 0, \rho = 0.5$), Normal ($\mu = 0, \rho = 1$), Uniform $[0, 1)$ Gaussian mixture, Zipfian. The latter 4 datasets are built as done in [6], [7], [10], and [11]: Modulo ($k = 16$), Two Dups ($A[i] = i^8 + N/2 \mod N$), Eight Dups ($A[i] = i^2 + N/2 \mod N$) and Root Dups ($A[i] = I \mod \sqrt{N}$).

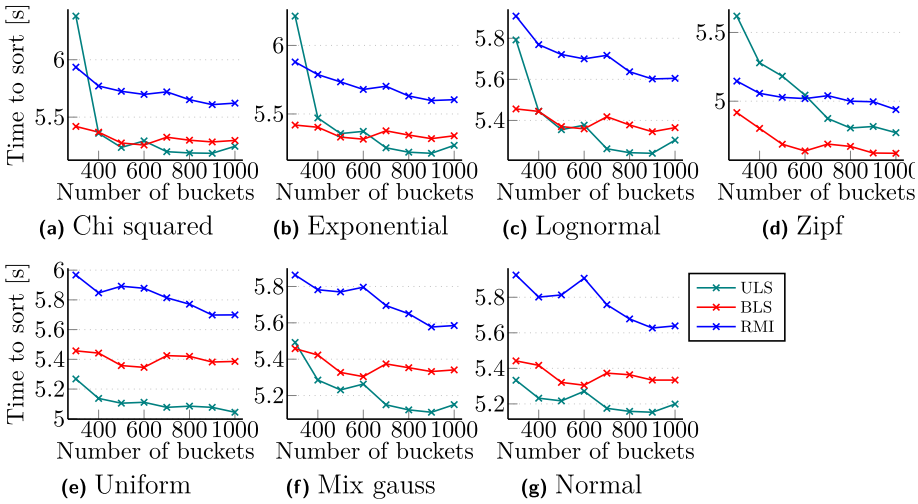


FIGURE 9. Sorting time (the lower the better) on 200 million of doubles drawn according to seven synthetic distributions.

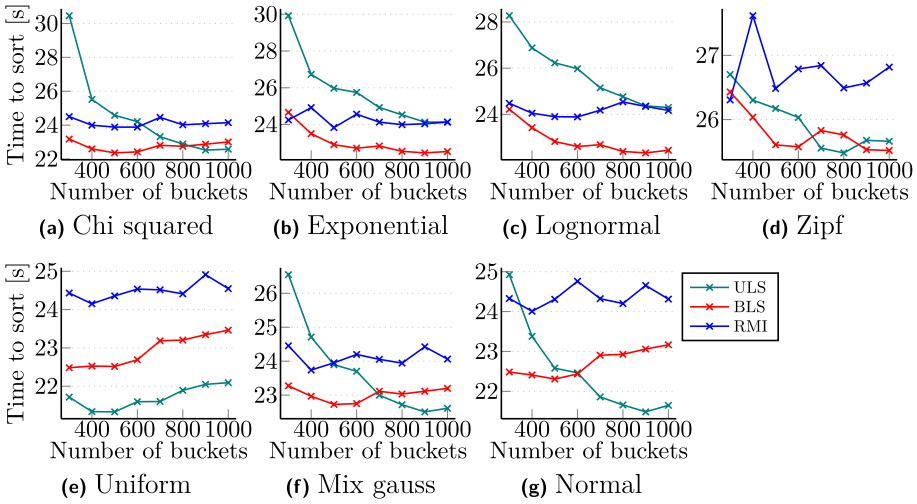


FIGURE 10. Sorting time (the lower the better) on 800 million of doubles drawn according to seven synthetic distributions.

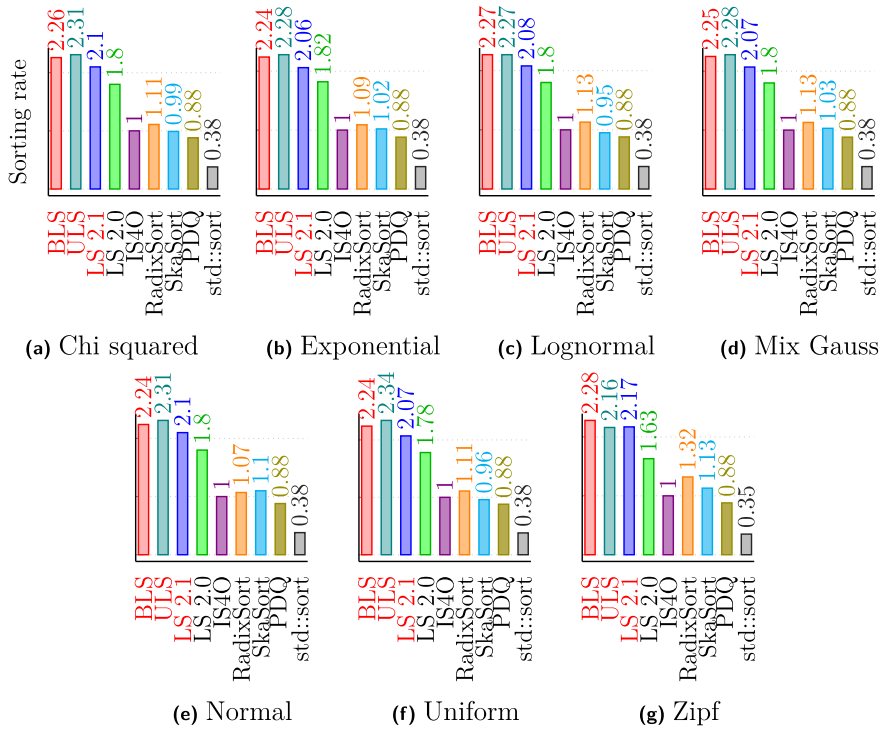


FIGURE 11. Sorting rate (the higher the better) on 200 million of doubles drawn according to seven synthetic distributions, expressed as the ratio with respect to the performance of IS^{4}o. In red are the names of the learned sorters proposed in this paper.

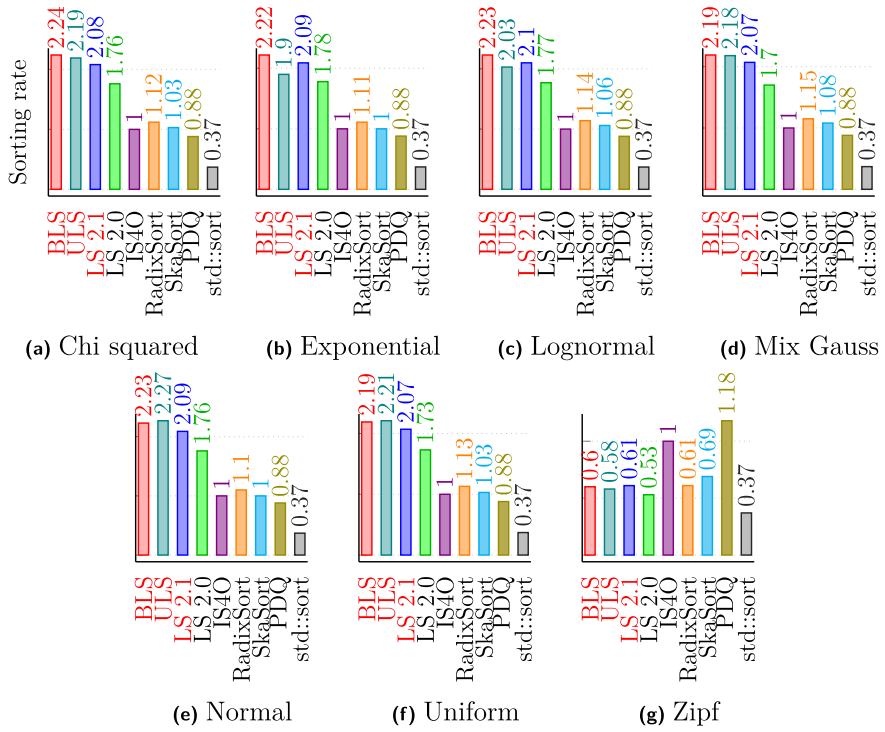
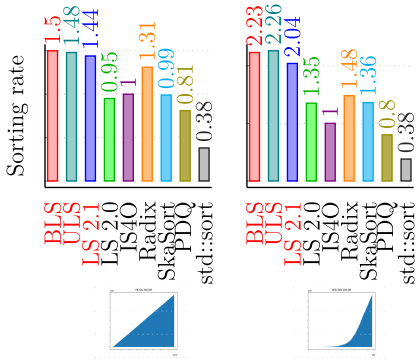


FIGURE 12. Sorting rate (the higher the better) on 800 million of doubles drawn according to seven synthetic distributions, expressed as the ratio with respect to the performance of IS^{4}o. In red are the names of the learned sorters proposed in this paper.

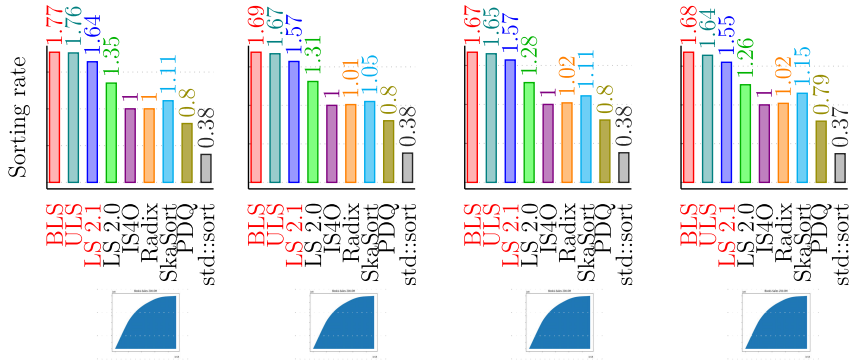
APPENDIX B
ALL REAL DATASETS: EXPERIMENTAL RESULTS

We experimented with nine sequential sorters, learned-based and classic, over a total of 22 real datasets of various sizes ranging from a few tens of millions to 800M items. These datasets have been used previously in the literature [7], [33], [34], [35], [36], [37], their description with all their corresponding experimental results are detailed in the caption of every following figure.



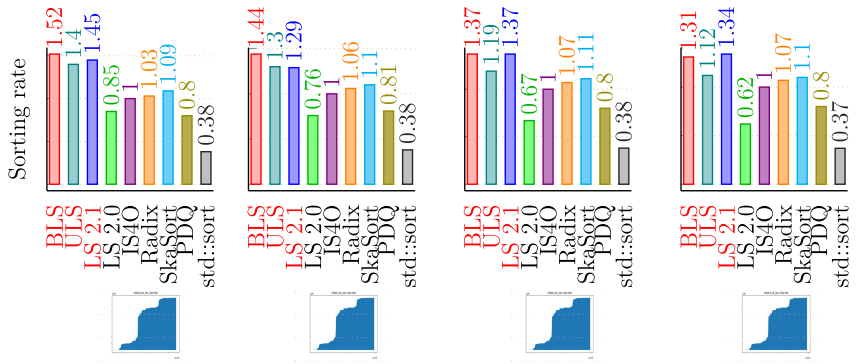
(a) FB (b) Wiki

FIGURE 13. The dataset on the left is a sample of 200M Facebook user IDs, obtained via a random walk of the user graph [33]. The distribution is almost uniform. As in [7], the outliers greater than the 0.99999 quantile are discarded. The second dataset on the right Wiki is drawn from Wikipedia and contains 200M article edit timestamps [33]. In both datasets all items are distinct.



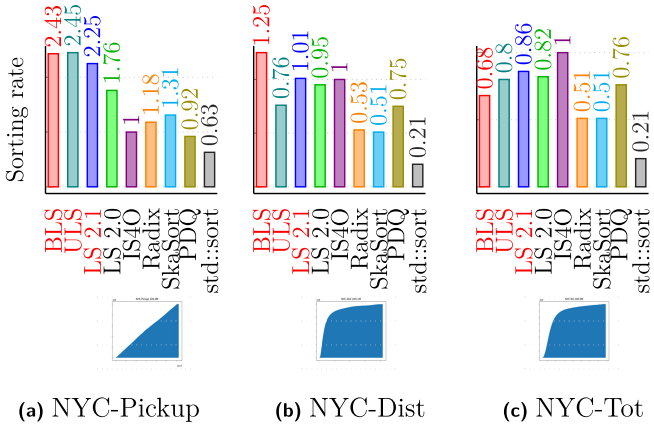
(a) Books 200M (b) Books 400M (c) Books 600M (d) Books 800M

FIGURE 14. Books dataset, with sizes ranging from 200M to 800M items. This dataset contains book sale popularity data from Amazon [33]. All items are distinct.



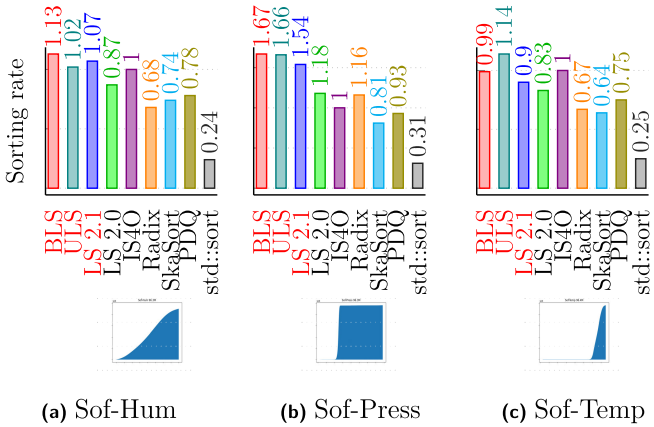
(a) Osm 200M (b) Osm 400M (c) Osm 600M (d) Osm 800M

FIGURE 15. The OpenStreetMaps dataset, with sizes ranging from 200M to 800M items, contains uniformly sampled location IDs from the entire world, represented as Google S2 Cell IDs [33]. This dataset has 45% of unique items.



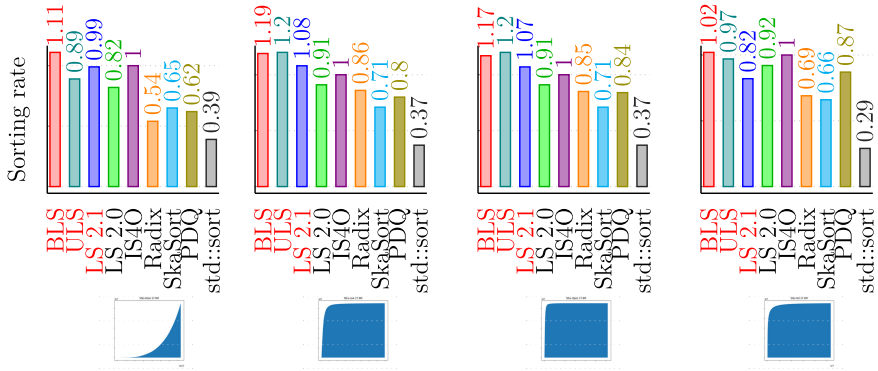
(a) NYC-Pickup (b) NYC-Dist (c) NYC-Tot

FIGURE 16. NYC dataset contains the yellow taxi trip records including fields that capture pick-up date-times, trip distances, and total fare [34]. They have sizes 100M, 200M, and 200M respectively. The unique items are ∼26.6M, 2060, and 7428 respectively.



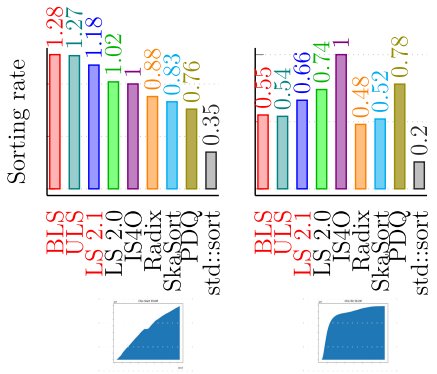
(a) Sof-Hum (b) Sof-Press (c) Sof-Temp

FIGURE 17. The Sofia dataset contains time-series air quality metrics measured from outdoor sensors in Sofia, Bulgaria [35]. All these datasets have a size of ∼96M real items, with a low number of unique items, respectively: 0.01%, 0.9%, and 0.01%.



(a) Stks-Date (b) Stks-Low (c) Stks-Open (d) Stks-Vol

FIGURE 18. Stks - Vol, Open, Low, and Date: this dataset contains historical daily opening prices for tickers trading on NASDAQ (stocks and ETFs), trading volumes, and trading dates [36]. Dataset size: ∼27.7M of records each. Percentage of unique items, respectively: 1.18%, 3.01%, 3.13%, 0.05%.



(a) Chic-Start (b) Chic-Tot

FIGURE 19. The Chicago Taxi Trips dataset contains taxi trips reported to the City of Chicago. The data to be sorted represents the trip starting timestamp and total fare [37]. The records are ∼39M, with 0.55% and 0.02% unique items.

ACKNOWLEDGMENT

The authors would like to thank Peter Sanders for his insights on the sorting problem and SOTAs, Florian Kurpicz for his invaluable support in setting up the benchmarks, and Maurizio Davini and the IT personnel with the Green Data Center of the University of Pisa for providing them with computing facilities.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming*, vol. 3. Reading, MA, USA: Addison-Wesley, 1973.

[2] I. Sabek and T. Kraska, "The case for learned in-memory joins," *Proc. VLDB Endowment*, vol. 16, no. 7, pp. 1749–1762, Mar. 2023.

[3] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[4] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "In-place parallel super scalar samplesort (IPSSSSo)," in *Proc. 25th Annu. Eur. Symp. Algorithms*, vol. 87, Vienna, Austria, Kirk Pruhs and Christian Sohler, Eds., 2017, pp. 1–14.

[5] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. Manage. Data*, May 2018, pp. 489–504.

[6] A. Kristo, K. Vaidya, U. Çetintemel, S. Misra, and T. Kraska, "The case for a learned sorting algorithm," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, Jun. 2020, pp. 1001–1016.

[7] A. Kristo, K. Vaidya, and T. Kraska, "Defeating duplicates: A re-design of the learnedsort algorithm," 2021, *arXiv:2107.03290*.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed., Cambridge, MA, USA: MIT Press, 2009.

[9] K.-D. Neubert, "The FlashSort1 algorithm," *Dr. Dobb's J.*, vol. 23, no. 2, pp. 123–129, 1998.

[10] I. Carvalho and R. Lawrence, "LearnedSort as a learning-augmented SampleSort: Analysis and parallelization," in *Proc. 35th Int. Conf. Scientific Stat. Database Manage.*, Los Angeles, CA, USA, R. Schuler, C. Kesselman, K. Chard, and A. Bugacov, Eds., Jul. 2023, pp. 1–9.

[11] M. Axtmann, S. Witt, D. Ferizovic, and P. Sanders, "Engineering in-place (shared-memory) sorting algorithms," *ACM Trans. Parallel Comput.*, vol. 9, no. 1, pp. 1–62, Jan. 2022.

[12] M. Athanassoulis, S. Idreos, and D. Shasha, "Data structures for data-intensive applications: Tradeoffs and design guidelines," *Found. Trends Databases*, vol. 13, nos. 1–2, pp. 1–168, 2023.

[13] P. Ferragina and G. Vinciguerra, "Learned data structures," in *Recent Trends in Learning From Data* (Recent Trends in Learning From Data), vol. 896, L. Oneto, N. Navarin, A. Sperduti, and D. Anguita, Eds., Berlin, Germany: Springer, 2019, pp. 5–41.

[14] P. Ferragina, F. Lillo, and G. Vinciguerra, "On the performance of learned data structures," *Theor. Comput. Sci.*, vol. 871, pp. 107–120, Jun. 2021.

[15] S. Zeighami and C. Shahabi, "On distribution dependent sub-logarithmic query time of learned indexing," in *Proc. Int. Conf. Mach. Learn. (ICML)*, Jan. 2023, pp. 40669–40680.

[16] J. Ding, U. F. Minhas, H. Zhang, Y. Li, C. Wang, B. Chandramouli, J. Gehrke, D. Kossmann, and D. B. Lomet, "ALEX: An updatable adaptive learned index," 2019, *arXiv:1905.08898*.

[17] A. Galakatos, M. Markovitch, C. Binnig, R. Fonseca, and T. Kraska, "FITing-tree: A data-aware index structure," in *Proc. Int. Conf. Manage. Data*, New York, NY, USA, Jun. 2019, pp. 1189–1206.

[18] P. Ferragina and G. Vinciguerra, "The PGM-index: A fully-dynamic compressed learned index with provable worst-case bounds," *Proc. VLDB Endowment*, vol. 13, no. 8, pp. 1162–1175, Apr. 2020.

[19] S. Chockchowwat, W. Liu, and Y. Park, "AirIndex: Versatile index tuning through data and storage," *Proc. ACM Manage. Data*, vol. 1, no. 3, pp. 1–26, Nov. 2023.

[20] J. Zhang and Y. Gao, "CARMI: A cache-aware learned index with a cost-based construction algorithm," *Proc. VLDB Endowment*, vol. 15, no. 11, pp. 2679–2691, Jul. 2022.

[21] A. Al-Mamun, H. Wu, Q. He, J. Wang, and W. G. Aref, "A survey of learned indexes for the multi-dimensional space," 2024, *arXiv:2403.06456*.

[22] Z. Sun, X. Zhou, and G. Li, "Learned index: A comprehensive experimental evaluation," *Proc. VLDB Endowment*, vol. 16, no. 8, pp. 1992–2004, Apr. 2023.

[23] S. Chatterjee, M. F. Pekala, L. Kruglyak, and S. Idreos, "Limousine: Blending learned and classical indexes to self-design larger-than-memory cloud storage engines," *Proc. ACM Manage. Data*, vol. 2, no. 1, pp. 1–28, Mar. 2024.

[24] A. Kristo and T. Kraska, "Parallel external sorting of ASCII records using learned models," 2023, *arXiv:2305.05671*.

[25] D. Amato, G. Lo Bosco, and R. Giancarlo, "Standard versus uniform binary search and their variants in learned static indexing: The case of the searching on sorted data benchmarking software platform," *Softw., Pract. Exper.*, vol. 53, no. 2, pp. 318–346, Feb. 2023.

[26] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz, "Practical massively parallel sorting," in *Proc. 27th ACM Symp. Parallelism Algorithms Architectures*, New York, NY, USA, Jun. 2015, pp. 13–23.

[27] P. Ferragina, *Pearls of Algorithm Engineering*. Cambridge, U.K.: Cambridge Univ. Press, 2023.

[28] L. A. Adamic and B. A. Huberman, "Zipf's law and the internet," *Glottometrics*, vol. 3, no. 1, pp. 143–150, 2002.

[29] O. R. L. Peters, "Pattern-defeating quicksort," 2021, *arXiv:2106.05123*.

[30] M. Skarupke. *I Wrote a Faster Sorting Algorithm*. Accessed: Apr. 10, 2024. [Online]. Available: https://probablydance.com/2016/12/27/i-wrote-a-faster-sorting-algorithm/

[31] *Radix Sort*. Accessed: Apr. 10, 2024. [Online]. Available: https://github.com/anikristo/LearnedSort/tree/master/third_party/radix_sort
[32] *STD: Sort*. Accessed: Apr. 10, 2024. [Online]. Available: https://en.cppreference.com/w/cpp/algorithm/sort
[33] A. Kipf, R. Marcus, A. V. Renen, M. Stoian, A. Kemper, T. Kraska, and T. Neumann, "SOSD: A benchmark for learned indexes," 2019, *arXiv:1911.13014*.
[34] *Tlc Trip Record Data*. Accessed: Apr. 10, 2024. [Online]. Available: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/SSDV7O
[35] *Sofia Weather Dataset*. Accessed: Apr. 10, 2024. [Online]. Available: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/WIOOVZ
[36] *Stocks Dataset*. Accessed: Apr. 10, 2024. [Online]. Available: https://dataverse.harvard.edu/dataset.xhtml?persistentId=doi:10.7910/DVN/8EOXQF
[37] *Chicago Taxi Trips Dataset*. Accessed: Apr. 10, 2024. [Online]. Available: https://data.cityofchicago.org/Transportation/Taxi-Trips/wrvz-psew#column-menu

PAOLO FERRAGINA received the Ph.D. degree in computer science from the University of Pisa, in 1996. He was a Postdoctoral Researcher with the Max-Planck Institut fuür Informatik, Saabruücken, Germany, in 1997 and 1998. He is currently a Professor in algorithms with the Scuola Superiore Sant'Anna and the University of Pisa, where he founded and leads the Acube Laboratory, whose research activities regard the design of algorithms for big data, mainly in the form of texts and graphs, in collaboration with companies worldwide: such as, e.g., Bloomberg, European Broadcasting Union (EBU), Google, Tiscali, and Yahoo! He has co-authored more than 190 (refereed) publications, four books, and several chapters, achieving a H-index of 38 and more than 7k citations on Scopus (43 and 11k on Google Scholar). His research results got five U.S./ITA patents and some international awards: e.g., "1995 Best Land Transportation Paper Award" from IEEE Vehicular Technology Society; "1997 Best Ph.D. Thesis in Theoretical Computer Science" by the Italian Chapter of the EATCS; "1997 Philip Morris Award on Science and Technology;" one Yahoo! Research Faculty Award; three Google Research Awards; and finally the "2022 ACM Paris Kanellakis Award." He is serving on the editorial board for the *Journal of Graph Algorithms and Applications* (JGAA). He is an Area Editor of two *Encyclopedia of Algorithms* (Springer) and *Big Data Technologies* (Springer).

MATTIA ODORISIO received the B.S. and M.S. degrees in computer science from the University of Pisa, where he is currently pursuing the Ph.D. degree. His research interests include sorting and learned data structures for big data applications.