The Case for a Learned Sorting Algorithm

Ani Kristo[*]
ani@brown.edu
Brown University

Kapil Vaidya[*]
kapilv@mit.edu
MIT

Uğur Çetintemel
ugur@cs.brown.edu
Brown University

Sanchit Misra
sanchit.misra@intel.com
Intel Labs

Tim Kraska
kraska@mit.edu
MIT

ABSTRACT

Sorting is one of the most fundamental algorithms in Computer Science and a common operation in databases not just for sorting query results but also as part of joins (i.e., sort-merge-join) or indexing. In this work, we introduce a new type of distribution sort that leverages a learned model of the empirical CDF of the data. Our algorithm uses a model to efficiently get an approximation of the scaled empirical CDF for each record key and map it to the corresponding position in the output array. We then apply a deterministic sorting algorithm that works well on nearly-sorted arrays (e.g., Insertion Sort) to establish a totally sorted order.

We compared this algorithm against common sorting approaches and measured its performance for up to 1 billion normally-distributed double-precision keys. The results show that our approach yields an average 3.38× performance improvement over C++ STL sort, which is an optimized Quicksort hybrid, 1.49× improvement over sequential Radix Sort, and 5.54× improvement over a C++ implementation of Timsort, which is the default sorting function for Java and Python.

ACM Reference Format:
Ani Kristo, Kapil Vaidya, Uğur Çetintemel, Sanchit Misra, and Tim Kraska. 2020. The Case for a Learned Sorting Algorithm. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389752

[*]Both authors contributed equally to this research.

1 INTRODUCTION

Sorting is one of the most fundamental and well-studied problems in Computer Science. Counting-based sorting algorithms, such as Radix Sort, have a complexity of $O(wN)$ with N being the number of keys and w being the key length and are often the fastest algorithms for small keys. However, for larger key domains comparison-based sorting algorithms are often faster, such as Quicksort or Mergesort, which have a time complexity of $O(N \log N)$, or hybrid algorithms, which combine various comparative and distributive sorting techniques. Those are also the default sorting algorithms used in most standard libraries (i.e., C++ STL sort).

In this paper, we introduce a ML-enhanced sorting algorithm by building on our previous work [28]. The core idea of the algorithm is simple: we train a CDF model F over a small sample of keys A and then use the model to predict the position of each key in the sorted output. If we would be able to train the perfect model of the empirical CDF, we could use the predicted probability $P(A \leq x)$ for a key x, scaled to the number of keys N, to predict the final position for every key in the sorted output: $pos = F_A(x) \cdot N = P(A \leq x) \cdot N$. Assuming the model already exists, this would allow us to sort the data with only one pass over the input, in $O(N)$ time.

Obviously, several challenges exists with this approach. Most importantly, it is unlikely that we can build a perfect empirical model. Furthermore, state-of-the-art approaches to model the CDF, in particular NN, would be overly expensive to train and execute. More surprising though, even with a perfect model the sorting time might be slower than a highly optimized Radix Sort algorithm. Radix Sort can be implemented to only use sequential writes, whereas a naïve ML-enhanced sorting algorithm as the one we outlined in [28] creates a lot of random writes to place the data directly into its sorted order.

In this paper, we describe Learned Sort, a sequential ML-enhanced, sorting algorithm that overcomes these challenges. In addition, we introduce a fast training and inference algorithm for CDF modeling. This paper is the first in-depth study describing a cache-efficient ML-enhanced sorting algorithm, which does not suffer from the random access problem. Our

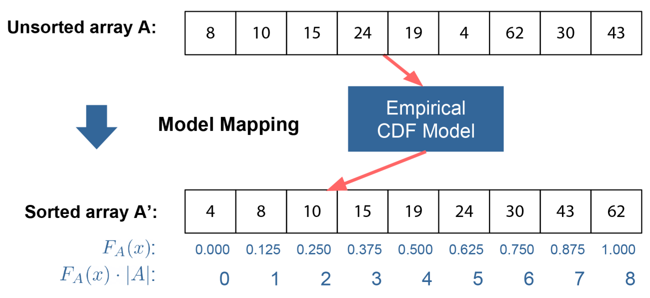

Figure 1: Sorting with the perfect CDF model

Algorithm 1 A first Learned Sort

```
    Input $A$ - the array to be sorted
    Input $F_A$ - the CDF model for the distribution of $A$
    Input $o$ - the over-allocation rate. Default=1
    Output $A'$ - the sorted version of array $A$
 1: procedure LEARNED-SORT($A$, $F_A$, $o$)
 2:     $N \leftarrow A$.length
 3:     $A' \leftarrow$ empty array of size $(N \cdot o)$
 4:     for $x$ in $A$ do
 5:         pos $\leftarrow \lfloor F_A(x) \cdot N \cdot o \rfloor$
 6:         if EMPTY($A'[pos]$) then $A'[pos] \leftarrow x$
 7:         else COLLISION-HANDLER($x$)
 8:     if $o > 1$ then COMPACT($A'$)
 9:     if NON-MONOTONIC then INSERTION-SORT($A'$)
10:     return $A'$
```

experiments show that Learned Sort can indeed achieve better performance than highly tuned counting-based sorting algorithms, including Radix Sort and histogram-based sorts, as well as comparison-based and hybrid sorting algorithms. In fact, our learned sorting algorithm provides the best performance even when we include the model training time as a part of the overall sorting time. For example, our experiments show that Learned Sort yields an average of 3.38× performance improvement over C++ STL sort (std::sort)[16], 5.54× improvement over Timsort (Python's default sorting algorithm [45]), 1.49× over Radix sort[51], and 1.31× over IS^{4}o[2], a cache-efficient version of the Samplesort and one of the fastest available sorting implementations [40].

In summary, we make the following contributions:

- We propose a first ML-enhanced sorting algorithm, called Learned Sort, which leverages simple ML models to model the empirical CDF to significantly speed-up a new variant of Radix Sort
- We theoretically analyze our sorting algorithm
- We exhaustively evaluate Learned Sort over various synthetic and real-world datasets

2 LEARNING TO SORT NUMBERS

Given a function $F_A(x)$, which returns the exact empirical CDF value for each key $x \in A$, we can sort A by calculating the position of each key within the sorted order as $pos \leftarrow F_A(x) \cdot |A|$. This would allow us to sort a dataset with a single pass over the data as visualized in Figure 1.

However, in general, we will not have a perfect CDF function, especially if we train the model just based on a sample from the input data. In addition, there might be duplicates in the dataset, which may cause several keys to be mapped to the same position. In the following, we describe an initial learned sorting algorithm, similar to the one of SageDB[28], that is robust against imprecise models, and then explain why this first approach is still not competitive, before introducing the final algorithm. To simplify the discussion, our focus in this section is exclusively on the sorting of numbers and we only describe the *out-of-place* variant of our algorithm, in which we use an auxiliary array as big as the input array. Later, we discuss the changes necessary to create an *in-place* variant of the same algorithm in Section 4.1 and address the sorting of strings and other complex objects in Section 4.2.

2.1 Sorting with imprecise models

As discussed earlier, duplicate keys and imprecise models may lead to the mapping of multiple keys to the same output position in the sorted array. Moreover, some models (e.g., NN or even the Recursive Model Index (RMI) [29]) may not be able to guarantee monotonicity, creating small misplacements in the output. That is, for two keys a and b with $a < b$ the CDF value of a might be greater than the one of b ($F(a) > F(b)$), thus, causing the output to not be entirely sorted. Obviously, such errors should be small as otherwise using a model would provide no benefits. However, if the model does not guarantee monotonicity, further work on the output is needed to repair such errors. That is a learned sorting algorithm also has to (1) correct the sort order for non-monotonic models, (2) handle key collisions, and preferably (3) minimize the number of such collisions. A general algorithm for dealing with those three issues is outlined in Algorithm 1. The core idea is again simple: given a model we calculate the expected position for each key (Line 5) and, if that position in the output is free, place the key there (Line 6). In case the position is not empty, we have several options to handle the collisions (Line 7):

(1) **Linear probing:** If a position is already occupied, we could sequentially scan the array for the nearest empty spot and place the element there. However, this technique might misplace keys (like non-monotonic models) and will take increasingly more time as the array fills up.
(2) **Chaining:** Like in hash-tables, we could chain elements for already-filled positions. This could be implemented either with a linked list or variable-sized sub-arrays, both of which introduce additional performance overhead due to pointer chasing and dynamic memory allocation.

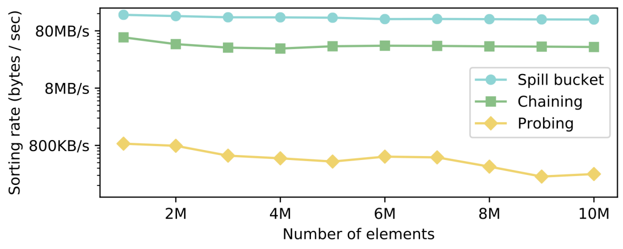

Figure 2: The sorting rate for different collision handling strategies for Algorithm 1 on normally distributed keys.

(3) **Spill bucket:** We could use a "spill bucket" where we append all the colliding elements. This technique requires to separately sort & merge the spill bucket.

We experimented with all three methods and found that the spill bucket often outperforms the other methods (see Figure 2). Thus, in the remainder of the paper, we only focus on the spill bucket approach.

After all items have been placed, for non-monotonic models we correct any mistakes using Insertion Sort over the entire output (Line 9). Note that, any sorting algorithm could guarantee correctness, however we choose Insertion Sort because it performs well when (1) the number of misplaced elements is small and (2) the distance of misplaced elements from their correct positions is small.

From Algorithm 1, it should be clear that the model quality determines the number of collisions and that the number of collisions will have a profound impact on the performance. Interestingly, the expected number of collisions depends on how well the model overfits to the observed data. For example, let us assume our CDF model is exactly the same model as used to generate the keys we want to sort (i.e., we have the model of the underlying distribution), the number of collisions would still be around $1/e \approx 36.7\%$ independently of the distribution. This result follows directly from the birthday paradox and is similar to the problem of hash table collision. However, if the model overfits to the observed data (i.e., learns the empirical CDF) the number of collisions can be significantly lower. Unfortunately, if we want to train a model just based on a sample, to reduce the training cost, it is mostly impossible to learn the perfect empirical CDF.

Hence, we need a different way to deal with collisions. For example, we can reduce the number of collisions by over-provisioning the output array (o in Algorithm 1), again similar to how hash tables are often over-provisioned to reduce collisions. However, this comes at the cost of requiring more memory and time to remove the empty space for the final output (Line 8). Another idea is to map keys to small buckets rather than individual positions. Bucketing helps significantly reduce the number of collisions and can be combined with over-allocation.

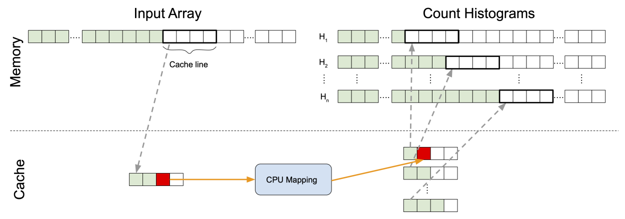

Figure 3: Radix Sort[51] can be implemented to mainly use sequential memory access by making sure that at least one cache line per histogram fits into the cache. This way the prefetcher can detect when to load the next cache-line per histogram (*green slots indicate processed items, red the current one, white slots unprocessed or empty slots*)

Algorithm 1 is in many ways similar to a hash table with a CDF model F_A as an order-preserving hash-function.[1] Yet, to our surprise, even with a perfect zero-overhead CDF model, Algorithm 1 is not faster than Radix Sort. For example, as a test we generated a randomly permuted dataset containing all integer numbers from 0 to 10^9. In this case, the key itself can be used as a position prediction as it represents the offset inside the sorted array, making the model for F_A just the identity function $pos \leftarrow key$; a perfect zero-overhead oracle. Note that we also maintain a bitmap to track if the positions are filled in the output array. To our astonishment, in this micro-experiment we observed that the time taken to distribute the keys into their final sorted position, despite a zero-overhead oracle function, took 38.7 sec and Radix Sort took 37.5 sec. This performance trend is due to the fact that this permutation step makes random and unpredictable array accesses, which hurt CPU cache and TLB's locality and incur multiple stalls (see Line 6 in Algorithm 1), whereas our cache-efficient Radix Sort implementation was memory-access optimized and mainly used sequential memory accesses[51]. The Radix Sort implementation achieved this by carefully adjusting the number of radices to the L2-cache size and while it made several passes over the data, it still outperformed our idealized learned sorting algorithm.

Based on the insights discussed in this section regarding random memory access, collision handling, and monotonicity, we developed a cache-efficient Learned Sort algorithm, which is explained in the next section.

2.2 Cache-optimized learned sorting

Our final Learned Sort algorithm enhances Algorithm 1 with the idea from the cache-optimized Radix Sort. In fact, in case the number of elements to sort is close to the key domain

[1]Note, that existing perfect or order-preserving hash-functions can not be used in our context because of their very slow training and execution time, which is also the reason why there does not exist a single sorting algorithm using them. Similarly our problem is NOT related to local-sensitive hashing either as sorting keys is in a single dimensional space.

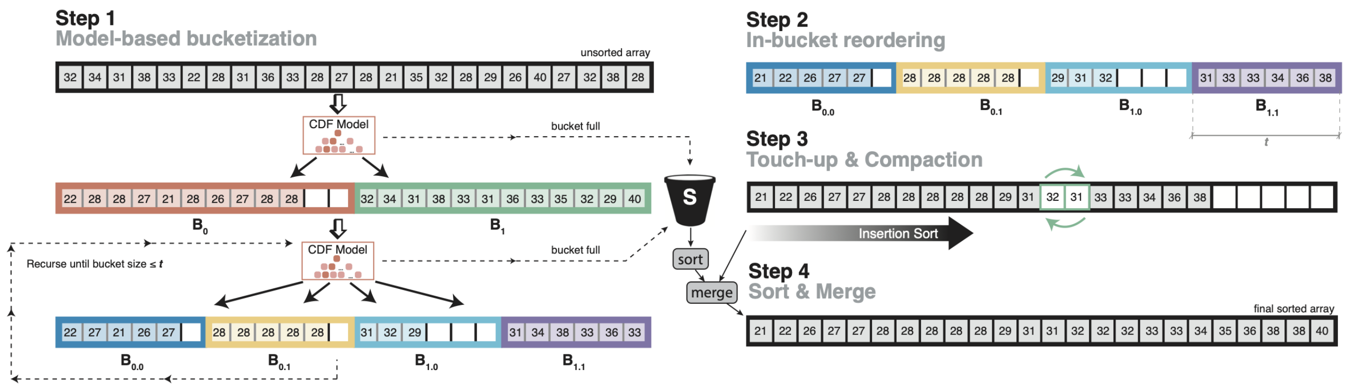

Figure 4: Cache-optimized Learned Sort: First the input is partitioned into f fixed-capacity buckets (here $f = 2$) and the input keys are shuffled into these buckets based on the CDF model's predictions. If a bucket gets full, the overflowing items are placed into a spill bucket S. Afterwards, each bucket is split again into f smaller buckets and the process repeats until the bucket capacity meets a threshold t (here $t = 6$). Then, each bucket is sorted using a CDF model-based counting sort-style subroutine (Step 2). The next step corrects any sorting mistakes using Insertion Sort (Step 3). Finally we sort the spill bucket S, merge it with B, and return the sorted array (Step 4).

size (e.g., 2^{32} for 32-bit keys) the run-time of our Learned Sort algorithm is almost identical to Radix Sort. However, in case the number of elements is much smaller than the key domain size, Learned Sort starts to significantly outperform even the optimized Radix Sort implementations as well as other comparison-based sorts. The reason is that, with every pass over the data our learned model can extract more information than Radix Sort about where the key should go in the final output, thus overcoming the core challenge of Radix Sort that the run-time heavily depends on the key domain size[2].

The basic idea of our algorithm is visualized in Figure 4:

- We organize the input array into logical buckets. That is, instead of predicting an exact position, the model only has to predict a *bucket index* for each element, which reduces the number of collisions as explained earlier.
- **Step 1:** For cache efficiency, we start with a few large buckets and recursively split them into smaller ones. By carefully choosing the fan-out (f) per iteration, we can ensure that at least one cache-line per bucket fits into the cache, hence transforming the memory access pattern into a more sequential one. This recursion repeats until the buckets become as small as a preset threshold t. Section 3.1 explains how f and t should be set based on the CPU cache size.
- **Step 2:** When the buckets reach capacity t, we use the CDF model to predict the *exact* position for each element within the bucket.
- **Step 3:** Afterwards we take the now sorted buckets and merge them into one sorted array. If we use a non-monotonic model, we also correct any sorting mistakes using Insertion Sort.
- **Step 4:** The buckets are of fixed capacity, which minimizes the cost of dynamic memory allocation. However, if a bucket becomes full, the additional keys are placed into a separate *spill bucket* array (see Figure 4 the "S"-bucket symbol). As a last step, the spill bucket has to be sorted and merged. The overhead of this operation is low as long as the model is capable of evenly distributing the keys to buckets.

Algorithm 2 shows the pseudocode of Learned Sort. The algorithm requires an input array of keys (A), a CDF model that was trained on a sample of this array (F_A), a fan-out factor (f) that determines the ratio of new buckets in each iteration, and a threshold (t) which decides when to stop the bucketization, such that every bucket fits into the cache.

Step 1: The algorithm starts by allocating a linear array B that is of the same size as the input A (Line 5). This will be *logically* partitioned into n buckets, each of fixed capacity b (Lines 3-4). We record the bucket sizes (i.e. how many elements are currently in the bucket) in an integer array I, which has the same size as the current number of buckets (n). Then, the algorithm shuffles each key into buckets by using the model F_A to predict its empirical CDF value and scaling it out to the current number of buckets in that round (Line 14). If the predicted bucket (at index pos) has reached its capacity, then the key is placed in the spill bucket S, otherwise, the key is inserted into the bucket (Lines 15 - 19). Here, we calculate the bucket start offset as $pos \cdot b$ and the write offset *within* the bucket as $I[pos]$. After one iteration, each bucket will be

[2]Obviously, the model itself still depends on the key domain size as discussed later in more detail.

logically split further into f smaller buckets (Lines 20-21) until the buckets are smaller than threshold t (Line 11). Note that, in order to preserve memory, we reuse the arrays A and B by simply swapping the read and write pointers (Line 22) and updating the bucket splitting parameters (Lines 20-21).

Step 2: When the bucket capacity reaches t, we switch to a model-based Counting Sort-style routine (Lines 23-38) to map the items to their final positions. We again do that using the model, which now predicts the *exact index position*, not the bucket. That is, we first calculate the final position for every key (Line 28) and store in array K the count of keys that are mapped to each predicted index (Line 29). The array K is then transformed to a running total (Line 31). Finally, we place the items into their final position using the cumulative counts (Lines 32-38), which is similar to the Counting Sort routine[7, pp.168-170]. As we only sort one bucket at a time and want to keep the array size of K small, we use an offset to set the start index of the bucket in Lines 28-36.

We switch to the model-based Counting Sort for two reasons. First, and most importantly, it helps improve the overall sorting time as we are able to fully utilize our model's precision for fine-level predictions. Second, it helps reduce the number of overflows (see Section 3.1.2 for more details).

Step 3: After the last sorting stage we remove any empty space and, for non-monotonic models, correct any potential mistakes with Insertion Sort(Line 40).

Step 4: Finally, because we used a spill bucket (S) for the overflowing elements in Stage 1, we have to sort it and merge it with the sorted buckets before returning (Lines 42-43). Provided that the sorting algorithm for the spill bucket is stable, Learned Sort also maintains the stability property.

2.3 Implementation optimizations

The pseudocode in Algorithm 2 gives an abstract view of the procedure, however, we have used a number of optimizations at the implementation level. Below we describe the most important ones:

- We process elements in batches. First we use the model to get the predicted indices for all the elements in the batch, and *then* place them into the predicted buckets. This batch-oriented approach maintains cache locality.
- As in Algorithm 1, we can over-provision array B by a small factor (e.g., 1.1×) in order to increase the bucket sizes and consequently reduce the number of overflowing elements in the spill bucket S. This in turn reduces the sorting time for S.
- Since the bucket sizes in Stage 2 are small (i.e., $b \leq t$), we can cache the predicted position for every element in the current bucket in Line 28 and reuse them in Line 34.
- In order to preserve the cache's and TLB's temporal locality, we use a bucket-at-a-time approach, where we perform all the operations in Lines 11-40 for all the keys in a single bucket before moving on to the next one.

The code for the algorithm can be found at http://dsg.csail.mit.edu/mlforsystems.

Algorithm 2 The Learned Sort algorithm

```
    Input $A$ - the array to be sorted
    Input $F_A$ - the CDF model for the distribution of $A$
    Input $f$ - fan-out of the algorithm
    Input $t$ - threshold for bucket size
    Output $A'$ - the sorted version of array $A$
 1: procedure LEARNED-SORT($A$, $F_A$, $f$, $t$)
 2:     $N \leftarrow |A|$                                  ▷ Size of the input array
 3:     $n \leftarrow f$                       ▷ $n$ represents the number of buckets
 4:     $b \leftarrow \lfloor N/f \rfloor$          ▷ $b$ represents the bucket capacity
 5:     $B \leftarrow [] \times N$                      ▷ Empty array of size $N$
 6:     $I \leftarrow [0] \times n$                       ▷ Records bucket sizes
 7:     $S \leftarrow []$                                          ▷ Spill bucket
 8:     read_arr ← pointer to $A$
 9:     write_arr ← pointer to $B$

10:     // Stage 1: Model-based bucketization
11:     while $b \geq t$ do      ▷ Until bucket capacity reaches the threshold $t$
12:         $I \leftarrow [0] \times n$                            ▷ Reset array I
13:         for $x \in$ read_arr do
14:             $pos \leftarrow \lfloor \text{INFER}(F_A, x) \cdot n \rfloor$
15:             if $I[pos] \geq b$ then                          ▷ Bucket is full
16:                 $S$.append($x$)                          ▷ Add to spill bucket
17:             else                         ▷ Write into the predicted bucket
18:                 write_arr[$pos \cdot b + I[pos]$] ← $x$
19:                 INCREMENT I[POS]
20:         $b \leftarrow \lfloor b/f \rfloor$                  ▷ Update bucket capacity
21:         $n \leftarrow \lfloor N/b \rfloor$             ▷ Update the number of buckets
22:         PTRSWP(read_arr, write_arr)         ▷ Pointer swap to reuse memory

23:     // Stage 2: In-bucket reordering
24:     offset ← 0
25:     for $i \leftarrow 0$ up to $n$ do                      ▷ Process each bucket
26:         $K \leftarrow [0] \times b$                           ▷ Array of counts

27:         for $j \leftarrow 0$ up to $I[i]$ do  ▷ Record the counts of the predicted positions
28:             $pos \leftarrow \lfloor \text{INFER}(F_A, \text{read\_arr}[\text{offset} + j]) \cdot N \rfloor$
29:             INCREMENT $K[pos - \text{offset}]$

30:         for $j \leftarrow 1$ up to $|K|$ do             ▷ Calculate the running total
31:             $K[j] \leftarrow K[j] + K[j-1]$

32:         $T \leftarrow []$                          ▷ Temporary auxiliary memory
33:         for $j \leftarrow 0$ up to $I[i]$ do  ▷ Order keys w.r.t. the cumulative counts
34:             $pos \leftarrow \lfloor \text{INFER}(F_A, \text{read\_arr}[\text{offset} + j]) \cdot N \rfloor$
35:             $T[j] \leftarrow \text{read\_arr}[\text{offset} + K[pos - \text{offset}]]$
36:             DECREMENT $K[pos - \text{offset}]$
37:         Copy $T$ back to read_arr[offset]
38:         offset ← offset + $b$

39:     // Stage 3: Touch-up
40:     INSERTION-SORT-AND-COMPACT(read_arr)

41:     // Stage 4: Sort & Merge
42:     SORT($S$)
43:     $A' \leftarrow$ MERGE(read_arr, $S$)

44:     return $A'$
```

2.4 Choice of the CDF model

Our sorting algorithm does not depend on a specific model to approximate the CDF. However, it is paramount that the model is fast to train and has a very low inference time

Algorithm 3 The inference procedure for the CDF model

```
**Input** $F_A$ - the trained model ($F_A[l][r]$ refers to the $r^{th}$ model in the $l^{th}$ layer)
**Input** $x$ - the key
**Output** $r$ - the predicted rank (between 0-1)
1: procedure INFER($F_A$, $x$)
2:     $L \leftarrow$ the number of layers of the CDF model $F_A$
3:     $M^l \leftarrow$ the number of models in the $l^{th}$ layer of the RMI $F_A$
4:     $r \leftarrow 0$
5:     for $l \leftarrow 0$ up to L do
6:         $r = x \cdot F_A[l][r]$.slope $+F_A[l][r]$.intercept
7:     return $r$
```

to keep the overall sorting time low. Thus, models such as KDE[43, 47], neural networks or even perfect order-preserving hash functions are usually too expensive to train or execute for our purposes. One might think that histograms would be an interesting alternative, and indeed histogram-based sorting algorithms have been proposed in the past [7, pp.168-177]. Unfortunately, histograms have the problem that they are either too coarse-grained, making any prediction very inaccurate, or too fine-grained, which increase the time to navigate the histogram itself (see also Section 6.8).

Certainly many model architectures could be used, however, for this paper we use the recursive model index (RMI) architecture as proposed in [29] (shown in Figure 5). RMIs contain simple linear models which are organized into a layered structure, acting like a mixture of experts[29].

2.4.1 Inference. Algorithm 3 shows the inference procedure for an RMI architecture. During inference, each layer of the model takes the key as an input and linearly transforms it to obtain a value, which is used as an index to pick a model in the next layer (Line 6). The intermediate models' slope an intercept terms are already scaled out to the number of models in the next layer, hence avoiding additional multiplications at inference time, whereas the last layer will return a CDF value between 0 and 1. Note, that the inference can be extremely fast because the procedure uses simple data dependencies instead of control dependencies (i.e., if-statements), consequently making it easier for the optimizer to perform loop unrolling and even vectorization. Hence, for each layer, the inference requires only one addition, one multiplication, and one array look-up to read the model parameters[29].

2.4.2 Training Procedure. Algorithm 4 shows the training procedure, which can be on a small sample of the input array. The algorithm starts by selecting a sample and sorting it using an efficient deterministic sorting algorithm – e.g., std::sort - (Lines 2-3), creating a 3D array to represent a tree structure of training sets, and inserting all <key, CDF> pairs into the the top node $T[0][0]$. Here the empirical CDF for the training tuples is calculated as its index in the sorted sample over the sample size ($i/|S|$). Starting at the root layer, the algorithm trains linear models working its way top-down.

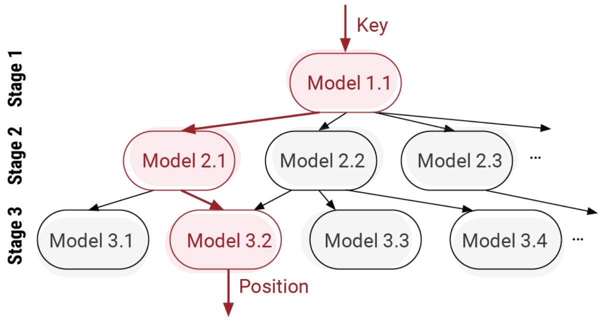

Figure 5: A typical RMI architecture containing three layers

Algorithm 4 The training procedure for the CDF model

```
**Input** $A$ - the input array
**Input** $L$ - the number of layers of the CDF model
**Input** $M^l$ - the number of linear models in the $l^{th}$ layer of the CDF model
**Output** $F_A$ - the trained CDF model with RMI architecture
1: procedure TRAIN($A$, $L$, $M$)
2:     $S \leftarrow$ SAMPLE(A)
3:     SORT($S$)
4:     $T \leftarrow [][][]$                    ▷ Training sets implemented as a 3D array
5:     for $i \leftarrow 0$ up to $|S|$ do
6:         $T[0][0]$.add(($S[i]$, $i/|S|$))
7:     for $l \leftarrow 0$ up to $L$ do
8:         for $m \leftarrow 0$ up to $M^l$ do
9:             $F_A[l][m] \leftarrow$linear model trained on the set $\{t \mid t \in T[l][m]\}$
10:            if $l + 1 < L$ then
11:                for $t \in T[l][m]$ do
12:                    $F_A[l][m]$.slope$\leftarrow F_A[l][m]$.slope $\cdot M^{l+1}$
13:                    $F_A[l][m]$.intercept$\leftarrow F_A[l][m]$.intercept $\cdot M^{l+1}$
14:                    $i \leftarrow F_A[l][m]$.slope $\cdot t + F_A[l][m]$.intercept
15:                    $T[l+1][i]$.add($t$)
16:    return $F_A$
```

The CDF model F_A can be implemented as a 2D array where $F_A[l][r]$ refers to the r^{th} model in the l^{th} layer of the RMI. For the root model, the algorithm uses the entire sample as training set to calculate a slope and intercept term (Line 9). After it has been trained, for each of the training tuples, the root model predicts a CDF value and it scales it by M^{l+1} (the number of models in the next layer) (Line 12-13). Then, it distributes these tuples into multiple training subsets that will be used to train each of the linear models in the subsequent layer. Each tuple goes to a training subset at index i, which is calculated in Line 14 by using the slope and intercept terms of the parent model. This partitioning process continues until the second-to-last layer of the RMI, and each of the newly created training subsets is used to train the corresponding linear models in a recursive fashion.

2.4.3 Training of the individual linear models. One way to train the linear models is using the closed-form of the univariate linear regression with an MSE loss function. However, when using linear regression training, it is possible that two neighboring linear models that are in the last layer of the CDF model predict values in overlapping ranges. Hence,

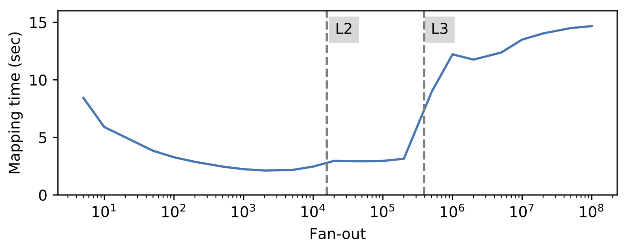

Figure 6: Mapping time for various fan-out values (log scale)

the resulting prediction is not guaranteed to be monotonic, increasing the time that Insertion Sort takes at the end of Algorithm 2. One way to force the CDF model to be monotonic is to use boundary checks for the prediction ranges of each leaf model, however, this comes at the expense of additional branching instructions for every element and at every iteration of the loop in Line 10 of Algorithm 2.

Thus, we opted instead to train the models using linear spline fitting which has better monotonicity. Furthermore, this method is cheaper to compute than the closed-form of linear regression and only requires to fit a line between the `min` and `max` key in the respective training sets.

From the perspective of a single model, splines seem to fit much worse than other models. However, recall that we use several linear models as part of the RMI, which overall maintains a good level of approximation of the CDF even for highly skewed distributions. Therefore, in contrast to [29], using linear splines, provides bigger advantages because (1) it is on average 2.7× faster to "train" than the closed-form version of linear regression, and (2) it provides up to 35% less key swaps during Insertion Sort.

3 ALGORITHM ANALYSIS

In this section we analyze the complexity and the performance of the algorithm w.r.t. various parameters.

3.1 Analysis of the sorting parameters

3.1.1 Choosing the fan-out (f). One key parameter of our algorithm is the fan-out f. On one hand, we want to have a high fan-out, as it allows us to leverage the model's accuracy to the fullest extent. On the other hand, in order to utilize the cache in the optimal way, we have to restrict the number of buckets, as we otherwise can not ensure that we can append to a bucket without causing a cache miss. Ideally, for all buckets we would keep in cache the auxiliary data structures, as well as the next empty slot per bucket.

In order to understand this trade-off, we measured the performance of Learned Sort with varying fan-out values (Figure 6) using a random array of 100M doubles, which is large enough to not fit in any cache level. The minimum of this plot corresponds to the point where all the hot memory locations fit in the L2 cache (f ≈1-5K). Empirically, the fan-out value that gives the best trade-off for the particular cache size in this setup is 1K. Hence, like in the cache-efficient Radix-Sort implementation, this parameter has to be tuned based on the available cache size.

3.1.2 Choosing the threshold (t). The threshold t determines the minimum bucket capacity as well as when to switch to a Counting Sort subroutine (Line 11 in Algorithm 2). We do this for two reasons: (1) to reduce the number of overflows (i.e., the number of elements in the spill bucket) and (2) to take better advantage of the model for the in-bucket sorting. Here we show how the threshold t affects the size of the spill bucket, which directly influences the performance.

If we had a perfect model every element would be mapped to a unique position. Yet, in most cases, this is impossible to achieve as we train based on a sample and aim to restrict the complexity of the model itself, inevitably mapping different items to the same position. Then our problem becomes that of randomly hashing N elements onto N unit-capacity buckets (i.e., $t = 1$). That is, the model that we learn behaves similar to an order-preserving hash function as a randomly generated element from this distribution is *equally* likely to be mapped onto any bucket. This holds for any distribution of the input array A, since its CDF function F_A) will be uniformly distributed between $[0, 1]$ [13]. Since we are using N buckets, the k^{th} bucket will contain the value range $[(k-1)/N, k/N)$ and the probability of the k^{th} bucket being empty is $(1 - 1/N)^N$, which is approximately $1/e$ for large N. This means that approximately N/e buckets will be empty at the end of the mapping phase, and all of these elements will be placed in the spill bucket. Using s to denote the size of the spill bucket, we have $\mathbf{E}[s] = N/e$.

In the general case, our problem is that of randomly hashing N elements into N/t buckets, each with capacity $t \geq 1$. Then, the expected number of overflowing elements is:

$$\mathbf{E}[s] = \frac{N}{te^t} \sum_{i=0}^{t-1} \frac{(t-i) \cdot (t)^i}{i!}$$

Capacity	Overflow	Capacity	Overflow
1	36.7%	25	7.9%
2	27.1%	50	5.6%
5	17.5%	100	3.9%
10	12.5%	1000	1.3%

Table 1: Bucket capacity (t) vs. proportion of elements in the spill bucket ($\mathbf{E}[s]/N$).

Table 1 represents the proportion of the elements in the spill bucket for various bucket capacities. Empirically, we found that we can maximize the performance of Learned Sort when the spill bucket contains less than 5% of the elements, therefore, we set the minimum bucket capacity to be $t = 100$.

3.1.3 The effect of model quality on the spill bucket size. The formula that we presented in the section above shows the expected size of the spill bucket when the model *perfectly* learns the underlying distribution. However, in this section, we analyze the expected size of the spill bucket for an approximation model that is trained on a small sample of the input.

For this analysis, we again assume that we can treat the sample as independently generated from the underlying distribution, and that we use unit-capacity buckets. For simplicity, we also linearly transform the input space into the range $[0, 1]$. Let $f(x)$ be the CDF of the underlying distribution and $g(x)$ be our approximation that is learned from the sample.

In the mapping phase, the keys in the range $[a, b]$ will be mapped to $N \cdot (g(b) - g(a))$ buckets. Whereas, in expectation, there will be $N \cdot (f(b) - f(a))$ keys present in the range $[a, b]$. The difference between the number of elements that are actually present in that range and the number of buckets that they are mapped to, leads to bucket overflows.

For a small range of input keys dx, the number of elements in the array within this input range $[x, x + dx)$ will be proportional to $f'(x)dx$ but our approximation will map them to $g'(x)dx$ buckets. So the problem turns into hashing $f'(x)dx$ elements into $g'(x)dx$ buckets of unit capacity. The number of empty buckets, which is same as number of overflowing elements, in this input range will be $g'(x)e^{-f'(x)/g'(x)}dx$, which will be integrated over the input range [0,1]. This gives us the following equation for measuring the number of overflowing elements w.r.t. the model's quality of approximation:

$$\mathbf{E}[s] = N \cdot \int_0^1 g'(x)e^{\frac{-f'(x)}{g'(x)}}\,dx$$

Using Jensen's inequality[25], it can be shown that this number is always greater than N/e with equality occurring when $f'(x) = g'(x)$. This shows that, for small samples, learning the underlying distribution leads to lesser elements in the spill bucket. A qualitative aspect of the formula above is that one needs to approximate the derivative of the CDF function (the PDF function) in order to minimize the expected size of spill bucket, and therefore maximize performance.

3.1.4 Choosing the sample size. We now discuss how the model quality changes w.r.t increasing sample size. We approximate the CDF of an element in a sample by looking at its position in the sorted sample. This empirical CDF of the sample is different from its CDF in the distribution that generated the sample. The Dvoretzky-Kiefer-Wolfowitz inequality[12] is one method for generating CDF-based confidence bounds and it states that for a given sample size n the difference between empirical CDF and real CDF is proportional to $O(1/\sqrt{n})$. So, as the sample size increases the accuracy of its empirical CDF improves making the model better.

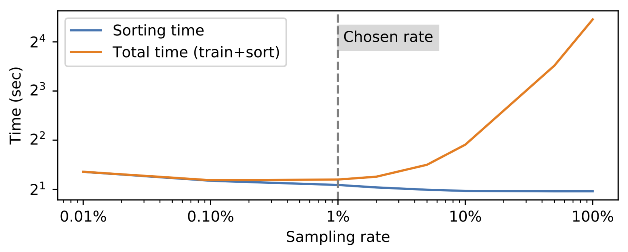

Figure 7: Sorting time for various sampling rates of 100M normally-distributed keys (log-log scale).

On other hand, a large sample increases the training time creating a trade-off between model quality and runtime performance. Figure 7 shows the trend of the time to sort the array (Sorting Time) and total time (Training and Sorting) w.r.t the sample size, while keeping the other parameters constant. In the figure, as sample size increases we can see the sorting time decreases because a larger sample leads to a better model quality. However, the training time keeps on increasing with the sample size. We empirically found that a sampling rate of 1% provides a good balance in this trade-off as is evident from the graph.

3.2 Complexity

Stage 1 of the sorting algorithm scans the input keys sequentially and for each key it uses the trained CDF model to predict which bucket to go to. This process takes $O(N \cdot L)$ time, where L is the number of layers in the model. Since we split the buckets progressively using a fan-out factor f until a threshold size t, the number of iterations and the actual complexity depend on the choice of f and t. However, in practice we use a large fan-out factor, therefore the number of iterations can be considered constant.

On the other hand, Stage 2 of the algorithm uses a routine similar to Counting Sort, which is a linear-time sorting procedure with respect to the bucket capacity (t), hence accounting for an $O(N)$ term. Assuming that the CDF model is monotonic, the worst case complexity of Stage 3 is $O(Nt)$ due to the fact that we use Insertion Sort and we have a threshold on the buckets. Otherwise, the worst case for a non-monotonic model would be $O(N^2)$. However, the constant term of this stage depends on the model quality: A good model, as the one described above, will provide a nearly-sorted array before the Insertion Sort subroutine is called, hence making this step *non-dominant* (refer to Figure 13).

Finally, as in the touch-up stage, the performance of Stage 4 also depends on the model quality. Assuming we employ a traditional, asymptotically optimal, comparison-based sorting routine for the spill bucket S, this stage's worst-case complexity is $O(s \log s) + O(N)$ ($O(N)$ for the merging step). Then again, a good model will not permit the size of the spill bucket to inflate (empirically $\leq 5\%$), which makes this step pretty insignificant in practice (see also Figure 13).

The space complexity of Learned Sort is in the order of $O(N)$, given that the algorithm uses an auxiliary memory for the buckets, which is linearly dependent on the input size. The in-place version introduced in Section 4.1, however, only uses a memory buffer that is independent of the input size, therefore, accounting for an $O(1)$ space complexity.

4 EXTENSIONS

So far we only discussed the Learned Sort implementation that is not in-place and does not handle strings or other complex objects. In this section, we provide an overview of how we can extend the Learned Sort algorithm to address these shortcoming, whereas in the Evaluation section we show experiments for an in-place version as well as early results for using Learned Sort over strings.

4.1 Making Learned Sort in-place

The current algorithm's space complexity is linear with respect to the input size due to the requirement from the mapping stage of the algorithm. However, there is a way of making the algorithm in-place, (i.e., have constant memory requirement that is independent on the input size).

The in-place version of the mapping stage would group the input keys into blocks such that all the elements in each block belong to the same bucket. The algorithm maintains a small buffer equal to the block size for every bucket. The algorithm iterates over the unsorted array and maps the elements into their respective buffer and whenever a buffer space fills up it is written onto the already scanned section of the array. These block are then permuted, so that blocks from the same bucket are stored contiguously. Note that this type of approach is very common for designing in-place algorithms such as in [2], and we show results in Section 6.6.

4.2 Learning to sort strings

The primary focus so far has been on sorting numerical values and extending our CDF model for strings creates a number of unique challenges. While we are still investigating on how to best handle strings and many existing work on ML-enhanced data structures and algorithms so far only considers numeric values [11, 29, 55], we outline an early implementation we did for strings, which also has a very compelling performance (see Section 6).

Our string model has an RMI architecture, but represents strings as input feature vectors $\mathbf{x} \in \mathbb{R}^n$ where n is the length of the string. For simplicity, we can work with fixed-length strings by padding shorter sequences with null characters. Then, one way to train the CDF model is to fit multivariate linear regression models ($\mathbf{w}.\mathbf{x}+\mathbf{b}$) over the feature vectors in each layer of the model. However, this strategy is computationally expensive as it would take $O(n)$ operations at every layer of the model. As a workaround, we could limit the characters considered by the model, however that might lead to non-monotonic predictions. If we consider $C_1, C_2, ..., C_n$ to be the ASCII values of characters in the string, then we can obtain a monotonic encoding of strings by calculating $\frac{C_1}{256} + \frac{C_2}{256^2} + .. + \frac{C_n}{256^n}$. This value is bound to be between zero and one, monotonic, and can potentially be used as a CDF value. This prediction would have been accurate if the ASCII value of each character was uniformly distributed and independent of the values of the other characters in the string. This is not always the case, so we transform the encodings to make their distribution uniform.

In the training phase, we take a sample from the array and encode the strings using their ASCII values and use them to map strings into the buckets. If the bucket sizes are uneven, we readjust the encoding ranges falling into these buckets by making a linear transformation of the slope and intercept terms of respective models. Then we re-map the strings into another layer of buckets after this linear transformation. This re-mapping step continues until we obtain evenly distributed buckets. Similar to the numeric algorithm we split the array into finer buckets until a threshold size after which point we use std::sort. Some promising preliminary results on this approach are shown in Section 6.4.

4.3 Duplicates

The number of duplicates (i.e., repeated keys) is a factor that affects the run-time behavior of our algorithm as our model will always assign the same bucket to the key, which, per definition, increases the number of collisions and the number of keys placed into the spill bucket. Consequently, the spill bucket inflates and *Stage 4* of the algorithm takes longer to execute, since it relies on a slower algorithm.

As a remedy, we incorporated a fast heuristic in our algorithm that detects repeated keys at training time. While building the CDF model, the algorithm looks at the frequencies at which equal keys appear in the training sample and, if it is above a certain threshold, it adds these keys to an *exception list*. Then, at sorting time, if the algorithm comes across an element whose key is in the exception list, it skips the bucket insertion step and only merges the repeated keys at the end of the procedure. However, in the absence of duplicates, we found that this additional step only introduces a small performance overhead (<3%), which is a tolerable cost for the average case.

5 RELATED WORK

Sorting algorithms are generally classified as *comparison-based* or *distribution-based*, depending on whether they rely only on pairwise key comparisons to come up with a sorted order, or rather make some assumptions or estimations on the distribution of the keys.

Comparison sorts: Some of the most common comparison sorts are Quicksort, Mergesort, and Insertion Sort. While they all have a lower bound of $\Omega(N \log N)$ comparisons in the average case, their performance in practice depends also largely on factors such as memory access patterns, which dictate their cache efficiency.

The GNU Standard Template Library in C++ employs Introsort [38] as its default sorting function (std::sort) [16], which combines the speed of Quicksort in the average case with the optimal worst case of Heapsort and the efficiency of Insertion Sort for small arrays. Introsort was also adopted by the standard library of the Go language[17] and that of the Swift language until version 5[1].

Samplesort is another variant of Quicksort that uses multiple pivots that are selected from a sample from the input array (note, that this does NOT create a histogram over the data). Thus, the elements are arranged into partitions in a finer-grained way that enables early termination of recursion when a certain partition contains only identical keys. One of the most recent and efficient implementation of Samplesort is the Super Scalar Samplesort initially introduced in [48] and then later on again with an in-place and improved implementation in [2] (IPS^{4}o). The strongest point of this implementation is the use of a binary search tree to efficiently discover the right partition for each key, and the avoidance of conditional branches with the help of conditional instructions provided in modern processors.

Java's List.sort() function[23] and Python's built-in sorting function[45] use a hybrid of Mergesort and Insertion Sort, called Timsort[36]. Timsort combines Insertion Sort's ability to benefit from the input's pre-sortedness with the stability of Mergesort, and it is said to work well on real-world data which contain intrinsic patterns. The procedure starts by scanning the input to find pre-sorted key sub-sequences and proceeds to merge them onto the final sorted output.

It should be noted, that most open-source DB systems implement their sorting routines building upon Quicksort or Mergesort, depending on whether the data fits in memory or if the ordering needs to be stable[37, 39, 46, 52].

Distribution sorts comprise the other major group of sorting procedures and they include algorithms like Radix Sort, Counting Sort, and Histogram Sort. Radix Sort is arguably the most commonly used one, and it works by calling the Counting Sort subroutine for all the keys by scanning a given number of digits at a time. The Counting Sort subroutine calculates a count histogram of all the keys based on the selected digits, transforms that into a cumulative histogram by generating running totals, and then re-orders the keys back into a sorted order based on these calculated counts. Radix Sort's time complexity is $O(d \cdot (N + r))$, where d is the number of passes over the data (i.e., the number of digits divided by the radix size), N is the input size, and r is the range of each key (i.e., 10 raised to the power of radix size). Note that, in order to use Radix Sort with IEEE-754 floating point numbers, it is first necessary to shift and mask the bit representation. While Radix Sort is highly sensitive to the key length, which dictates the number of passes, it is nevertheless a very efficient sorting algorithm for numerical types, that is very well-suited for multi-core procedures[6, 22, 40], and SIMD vectorization [50].

Most related to Learned Sort is Histogram Sort[4]. However, Histogram Sort implicitly assumes a uniform distribution for the input data as it allocates n variable-sized buckets and maps each key x into a bucket B_i by calculating $i = n \cdot (x - x_{min})/(x_{max} - x_{min})$. It then sorts these buckets using Insertion Sort and merge them in order.

SIMD optimization: There has been a lot of work on enhancing traditional sorting implementations with data parallelism in SIMD-enabled CPUs[5, 14, 26], as well as the use of adaptive and cache-sensitive partitioning techniques for multi-core or multi-processor implementations [3, 6, 9, 21, 50]. Nevertheless, there has not been much recent innovation in the algorithmic space for sorting and we found that IS^{4}o, one of our baselines, is one of the most competitive openly available implementations.

Hashing functions In a way, the CDF model might be regarded as an order-preserving hash function for the input keys, such as [8, 32]. However, order-preserving hashing is unsuitable for sorting since it does not provide fast enough training and inference times, and, to the best of our knowledge, there does not exist any sorting algorithm that uses order-preserving hashing for sorting. Similarly, locality-sensitive hashing [54, 56, 57] can also not be used for sorting a single numeric value as we are concerned with sorting a single dimension rather than efficiently finding similar items in a multi-dimensional space. Finally, perfect hashing's objective is to avoid element collisions, which would initially seem an interesting choice to use for Learned Sort. However, perfect hash functions grow in size with the input data, are not fast to train, and most importantly, usually not order-preserving [10].

ML-enhanced algorithms There has been growing interest in the use of Machine Learning techniques to speed up traditional algorithms in the context of systems. Most notably, the work in Learned Index Structures [29] introduces the use of an RMI structure to substitute the traditional

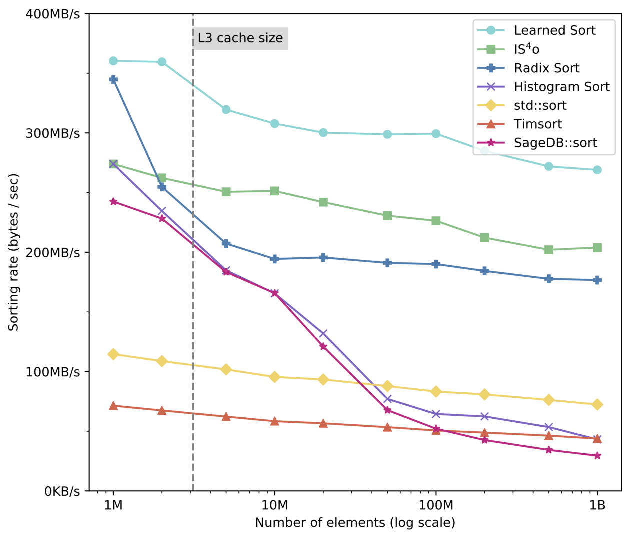

Figure 8: The sorting throughput for normally distributed double-precision keys (higher is better).

B+ tree indexes in database systems, a work that was followed up later on by [11]. In addition, other ML algorithms, as well as reinforcement learning, have been used to tune system parameters or optimize query execution plans in [27, 30, 34, 35, 58]. Finally, this new research trend has also reached other system applications outside databases, such as in scheduling [33], congestion control [24], and frequency estimation in data stream processing [20].

6 EVALUATION

The goal of this section is to:

- Evaluate the performance of Learned Sort compared to other highly-tuned sorting algorithms over real and synthetic datasets
- Explain the time-breakdown of the different stages of Learned Sort
- Evaluate the main weakness of Learned Sort: duplicates
- Show the relative performance of the in-place versus the out-of-place Learned Sort
- Evaluate the Learned Sort performance over strings.

6.1 Setup and datasets

As baselines, we compare against cache-optimized and highly tuned C++ implementations of Radix Sort [51], Timsort [18], Introsort (std::sort), Histogram Sort[4], and IS^{4}o [49] (one of the most optimized sorting algorithms we were able to find, which was also recently used in other studies [40] as a comparison point). Note that we use a recursive, equi-depth version of Histogram sort that adapts to the input's skew as to avoid severe performance penalties. While we are presenting only the most competitive baselines, we have, in fact, conducted measurements against Mergesort, Heapsort, Insertion Sort, Shell Sort, Quicksort, and one of its improved variants – PDQS[44]. However, we did not consider them further as their performance was always worse than one of the other baselines. Note, that for all our experiments **we include the model training time as part of the overall sorting time** unless mentioned otherwise.

We evaluate the performance of our algorithm for numerical keys on both synthetic and real datasets of varying precision. For the synthetic datasets we generated the following distributions:

- Uniform distribution with `min=0` and `max=1`
- Multimodal distribution that is a mixture of five normal distributions whose PDF is shown in the histogram below the performance charts in Figure 9
- Exponential distribution with $\lambda = 2$ and scaled by a factor of 10^6 (80% of the keys are concentrated in 7% of the key domain)
- Lognormal distribution with $\mu = 0$ and $\sigma = 1$ that has an *extreme skew* (80% of the keys are concentrated in 1% of the key range)

We also use real-world data from OpenStreetMap[42] and sort on 100M longitude and latitude compound keys (`osm/longlat`) that are generated using the transformation `longlat` $=$ `180` $\cdot$ `lon` $+$ `lat`, as in [11], as well as on their respective node IDs (`osm/id`). In addition, we use an IoT dataset [15] to sort on the `iot/mem` and `iot/bytes` columns (10M keys), which represent the amount of available memory and number of input bytes to a server machine at regular time intervals. Thirdly, we use the Facebook RW dataset (`fb/rw`) to sort on 1.1M collected user IDs from a random walk in the Facebook user graph[31]. Finally, we show results from TPC-H benchmark data on the customer account balances (`tpch/bal` - 3M keys), and order keys (`tpch/o_key` - 30M keys) for a scale factor of 20, which are of course not real but represents data which are often used to evaluate the performance of database systems. All datasets were randomly shuffled before sorting. We display the distribution of these datasets with histograms below each result in Figure 9.

All the experiments are measured on a server-grade Linux machine running on Intel® Xeon® Gold 6150 CPU @ 2.70GHz with 376GB of memory, and compiled with GCC 9.2 with the `-O3` flag for full optimizations[3]. The model we used for training was always 2-layers and contained 1000 leaf models, trained with a uniformly selected 1% sample of the array.

[3]Intel Xeon and Intel Xeon Phi are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Other names and brands may be claimed as the property of others. ©Intel Corporation.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change

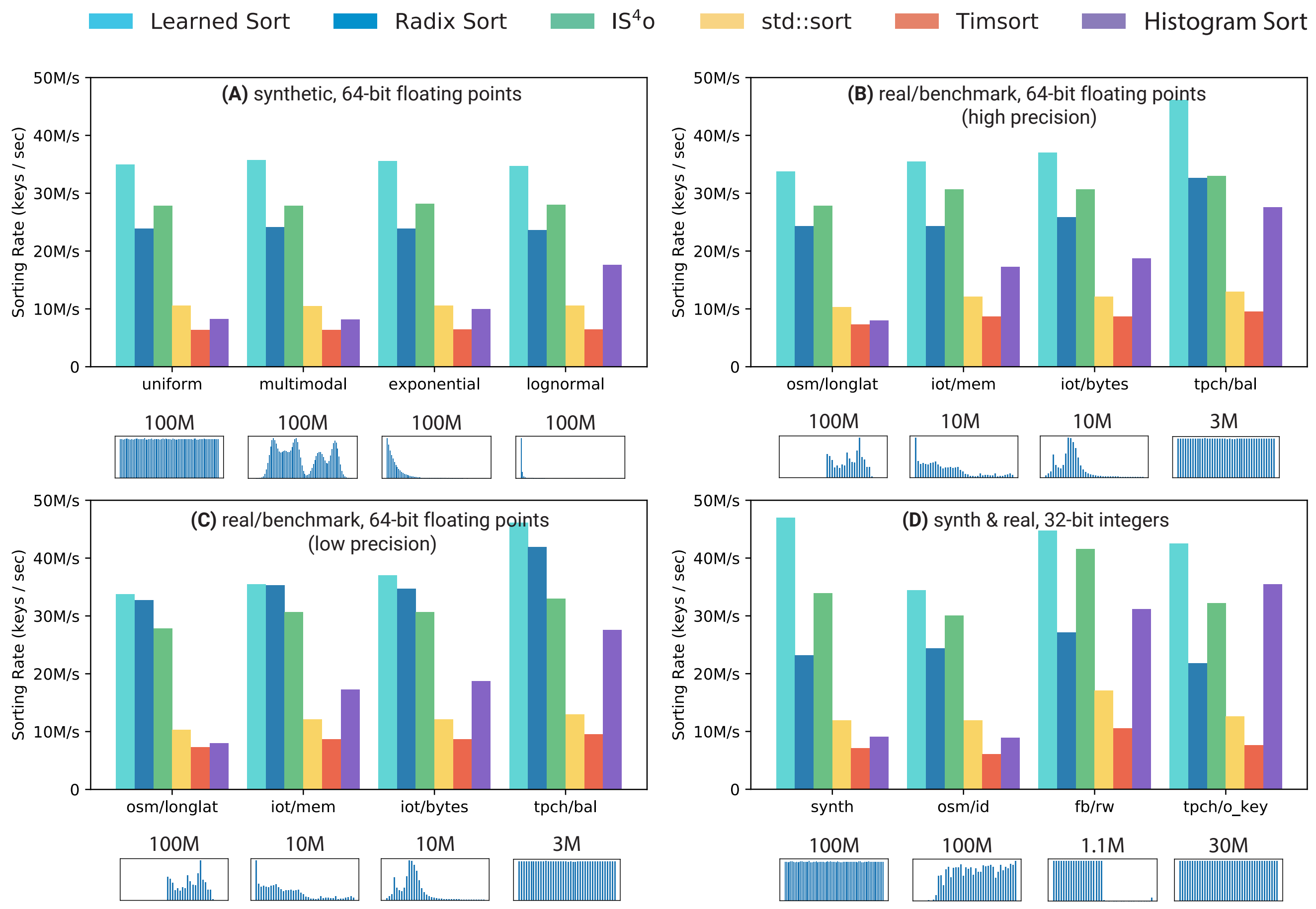

Figure 9: The sorting rate of Learned Sort and other baselines for real and synthetic datasets containing both doubles and integers. The pictures below the charts visualize the key distributions and the dataset sizes.

6.2 Overall Performance

As the first experiment we measured the sorting rate for array sizes varying from 1 million up to 1 billion double-precision keys following a standard normal distribution, and compare it to the baseline algorithms that we selected as described in Section 6.1. The sorting rate (bytes per second) is shown in Figure 8 for Learned Sort and our main baselines, in addition to SageDB::sort[28]. As it can be seen **Learned Sort achieves an average of 30% higher throughput** than the next best algorithm (IS^{4}o) and 55% as compared to Radix Sort for larger data sizes. However, when the data fits into the L3 cache, as it is the case with 1 million keys (roughly 8MB), Radix sort is almost as fast as Learned Sort. However, as soon as the data does not fit into the L3 cache, the sorting rate of Learned Sort is significantly higher than Radix or IS^{4}o. Furthermore, Learned Sort's cache optimization enables it to maintain a good sorting throughput even for sizes up to 8GB.

6.3 Sorting rate

To better understand the behaviour of our algorithm, we compared Learned Sort against our other baselines on (A) synthetic data with 64-bit doubles generated from different distributions, (B) high precision real-world/benchmark data with 64-bit doubles, which have at least 10 significant digits, (C) low precision real-world/benchmark data with 64-bit doubles, with reduced floating point precision, and (D) synthetic and real-world data with 32-bit integers.

Figure 9A shows that Learned Sort consistently outperforms Radix Sort by an average of 48%, IS^{4}o by an average

to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to www.intel.com/benchmarks.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

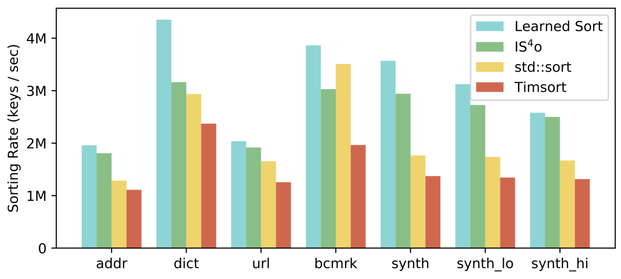

Figure 10: The sorting rate for various strings datasets.

of 27%, and the other baselines by much larger margins. The same performance gain is also present for the high-precision real-world datasets (Figure 9B). Note, that we achieve a significant higher sorting rate for the `tpch/bal` data due to the fact that it is smaller, and incurs more cache benefits.

On the other hand, we observed that Radix Sort's performance improves when the input keys have less precision (Figure 9C). In this case, Learned Sort and all the other baselines algorithms remain unaffected, while Radix Sort gets a 34% performance boost. This improvement results from the fact that everything can be sorted on the most significant bits. However, it is necessary to note that Radix Sort's performance does not surpass that of Learned Sort.

Finally, in Figure 9A we show that for integers Learned Sort has an even higher throughput and, in some cases, even a bigger benefit. For example, on the synthetic integer dataset it is 38% better than IS^{4}o and twice as fast as Radix Sort. Whereas for the FB dataset and OSM IDs the performance difference compared to IS^{4}o is less because of the particular distribution of values and duplicates (see Section 4.3).

6.4 Sorting Strings

Figure 10 show the preliminary sorting rate for strings for our algorithm of Section 4.2 with respect to IS^{4}o, std::sort, and Timsort. In this experiment we excluded the training time for the model. However, it should be noted, that many real-world scenarios exists in which a dataset or a subset has to be sorted several times. For example, within a database recurring merge-joins operation or the sorting for the final result, would allow to pre-train models as similar (but not identical) subsets of the data might appear over and over again. Note, that we excluded Radix Sort from this comparison as it was significantly slower than any of the other baselines. For the data we used:

- **addr**: A set of 1M address strings from the OpenAddresses dataset (Northeast USA)[41].
- **dict**: A set of 479K words from an English dictionary[53].
- **url**: A list of 1.1M URLs from the Weblogs dataset[15] containing requests to a webserver for *cs.brown.edu*
- **bcmrk**: 10M ASCII arrays generated using the code from the SortBenchmark dataset[19]
- **synth, synth_lo, synth_hi** : A set of 1M randomly generated strings following a uniform distribution of a-z characters at each position with characters having no correlation, low correlation, and high correlation with neighbouring ASCII values, respectively.

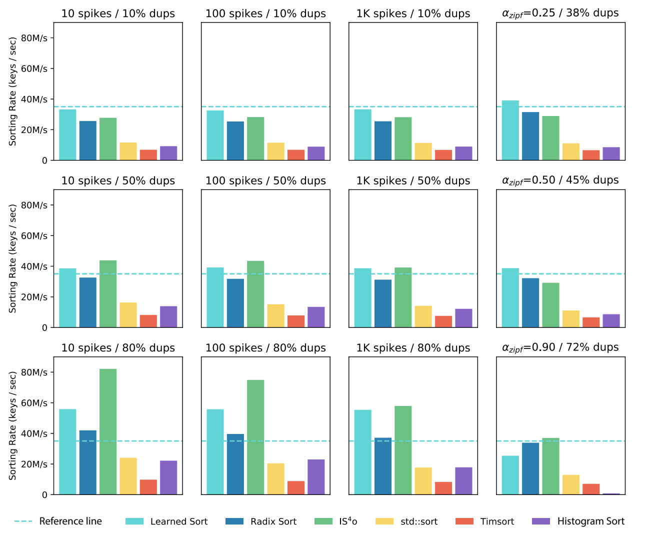

Figure 11: The sorting rate of Learned Sort and the other baselines for varying degrees of duplicated keys and number of spikes, as well as on different Zipf distributions. The reference line represents the sorting rate of Learned Sort where there are no duplicates.

Overall, the experiment show that Learned Sort is also a very promising direction for sorting complex objects, such as strings. It should be noted, that building efficient models for string is still an active area of research and probably a paper on its own as it also have far reaching applications for indexes, tries, and many other data structures and algorithms. Finally, we would like to point out that if we include the training time with the sorting time, Learned Sort still dominates the other algorithms but by a margin of of 2-8% rather than the 5-20% shown in Figure 9.

6.5 The impact of duplicates

The number of duplicates (i.e., repeated keys) is a factor that affects the run-time behavior of our algorithm as our model will always assign the same position/bucket to the key, which increases the number of collisions and the number of keys placed into the spill bucket. To study the impact of duplicates, we first generate a normal distribution dataset ($\mu = 0$ and $\sigma = 5$) and afterwards duplicate a few randomly selected keys n-times (referred to as spikes).

Figure 11 shows the performance of Learned Sort and the other baseline algorithms for different combinations of the

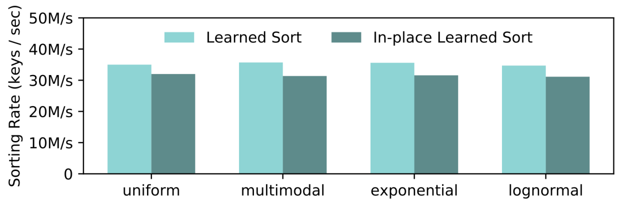

Figure 12: The sorting rate of Learned Sort and its in-place version for all of our synthetic datasets.

number of spikes and number of duplicate keys, in addition to three different Zipf distributions with parameters 0.25, 0.50, and 0.90. As the results demonstrate our technique from Section 4.3 ensures that Learned Sort remains highly competitive even for datasets with large degrees of duplicates. Only for the most extreme cases with 50-80% of duplicates IS^{4}o is actually faster than Learned Sort.

6.6 In-place sorting

The performance of the in-place version described in Section 4.1 is shown in Figure 12, and as observed, the sorting rate drops by an average of 8-11% as compared to when the mapping procedure directly uses fixed-capacity buckets.

6.7 Performance decomposition

Next, in Figure 13 we show the time spent in each phase of Learned Sort. With 1% sample from the input, the training procedure only accounts for $\approx$ 6% of the overall runtime. The majority of time ($\approx$ 80%) goes towards the model-based key bucketization. Since the CDF model makes a nearly-sorted order, the touch-up step with Insertion Sort makes up only 5% of the total time, and the spill bucket sorting only 2%, which reconfirms the quality of the model predictions.

6.8 Using histograms as CDF models

Finally, in Figure 14 we run a micro-experiment to show the performance of a version of our Learned Sort algorithm that uses a histogram as CDF model. We also compare this approach with Histogram Sort[4]. For both these algorithms we show the performance using both, equi-width and equi-depth histograms. The equi-depth CDF histogram model was implemented as a linear array that records the starting key for each bin and uses binary search to find the right bin for any given key. On the other hand, for the equi-width version we do not need to store the keys and we can find the right bin using a simple array lookup.

As a reference the figure includes the RMI-based Learned Sort as a dashed line at around 280MB/s. The figure shows that the number of bins has a clear impact on the sorting rate of the histogram based method. Surprisingly, the Histogram Sort using equi-depth performs better than using our Learned

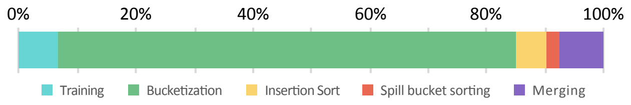

Figure 13: Performance of each of the stages of Learned Sort.

Sort with a histogram as a model. The reason is, that the additional number of passes over the data performed as part of Learned Sort does not pay out for the imprecision of the histogram-based models (consider, for example, Step 2 in Algorithm 2). However, Learned Sort with an RMI model is almost twice as fast. The advantage of using RMI models comes from the fact, that it uses continuous functions to quickly estimate the position for a key.

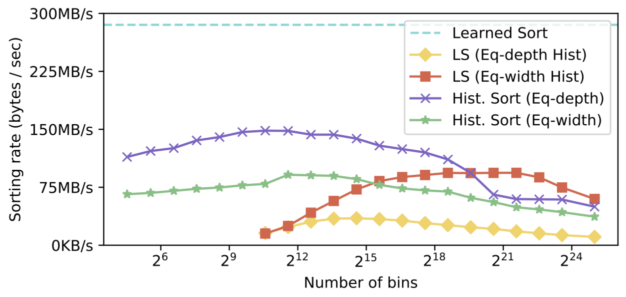

Figure 14: The sorting rate of Learned Sort algorithm on 100M normally-distributed keys as compared with (1) a version of LS that uses an equi-depth histogram as CDF model, (2) a version with an equi-width histogram, (3) Equi-depth Histogram Sort, and (4) Equi-width Histogram Sort.

7 CONCLUSION AND FUTURE WORK

In this paper we presented a novel approach to accelerate sorting by leveraging models that approximate the empirical CDF of the input to quickly map elements into the sorted position within a small error margin. This approach results in significant performance improvements as compared to the most competitive and widely used sorting algorithms, and marks an important step into building ML-enhanced algorithms and data structures. Much future work remains open, most notably how to handle strings, complex types, and parallel sorting.

ACKNOWLEDGMENTS

This research is supported by Google, Intel, and Microsoft as part of the MIT Data Systems and AI Lab (DSAIL) at MIT, NSF IIS 1900933, DARPA Award 16-43-D3M-FP040, and the MIT Air Force Artificial Intelligence Innovation Accelerator (AIIA).

REFERENCES

[1] Apple. 2018. Apple/Swift: standard library sort. (2018). https://github.com/apple/swift/blob/master/test/Prototypes/IntroSort.swift

[2] Michael Axtmann, Sascha Witt, Daniel Ferizovic, and Peter Sanders. 2017. In-Place Parallel Super Scalar Samplesort (IPSSSSo). In *25th Annual European Symposium on Algorithms (ESA 2017) (Leibniz International Proceedings in Informatics (LIPIcs))*, Kirk Pruhs and Christian Sohler (Eds.), Vol. 87. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 9:1–9:14. https://doi.org/10.4230/LIPIcs.ESA.2017.9

[3] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment* 7, 1 (2013), 85–96.

[4] Paul E. Black. 2019. Histogram Sort. (2019). https://www.nist.gov/dads/HTML/histogramSort.html

[5] Berenger Bramas. 2017. Fast sorting algorithms using AVX-512 on Intel Knights Landing. *arXiv preprint arXiv:1704.08579* 305 (2017), 315.

[6] Minsik Cho, Daniel Brand, Rajesh Bordawekar, Ulrich Finkler, Vincent Kulandaisamy, and Ruchir Puri. 2015. PARADIS: an efficient parallel algorithm for in-place radix sort. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1518–1529.

[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2007. *Introduction to algorithms* (2 ed.). MIT Press, Cambridge, MA.

[8] Zbigniew J. Czech, George Havas, and Bohdan S. Majewski. 1992. An optimal algorithm for generating minimal perfect hash functions. *Inform. Process. Lett.* 43, 5 (1992), 257 – 264. https://doi.org/10.1016/0020-0190(92)90220-P

[9] T. Dachraoui and L. Narayanan. 1996. Fast deterministic sorting on large parallel machines. In *Proceedings of SPDP '96: 8th IEEE Symposium on Parallel and Distributed Processing*. IEEE, New Orleans, LA, 273–280. https://doi.org/10.1109/SPDP.1996.570344

[10] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E Tarjan. 1994. Dynamic perfect hashing: Upper and lower bounds. *SIAM J. Comput.* 23, 4 (1994), 738–761.

[11] Jialin Ding, Umar Farooq Minhas, Hantian Zhang, Yinan Li, Chi Wang, Badrish Chandramouli, Johannes Gehrke, Donald Kossmann, and David Lomet. 2019. ALEX: An Updatable Adaptive Learned Index. (2019). arXiv:cs.DB/1905.08898

[12] A. Dvoretzky, J. Kiefer, and J. Wolfowitz. 1956. Asymptotic Minimax Character of the Sample Distribution Function and of the Classical Multinomial Estimator. *Ann. Math. Statist.* 27, 3 (09 1956), 642–669. https://doi.org/10.1214/aoms/1177728174

[13] Paul Embrechts and Marius Hofert. 2013. A note on generalized inverses. *Mathematical Methods of Operations Research* 77, 3 (2013), 423–432.

[14] Timothy Furtak, José Nelson Amaral, and Robert Niewiadomski. 2007. Using SIMD registers and instructions to enable instruction-level parallelism in sorting algorithms. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, San Diego, CA, 348–357.

[15] Alex Galakatos, Michael Markovitch, Carsten Binnig, Rodrigo Fonseca, and Tim Kraska. 2019. FITing-Tree: A Data-Aware Index Structure. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1189–1206. https://doi.org/10.1145/3299869.3319860

[16] GNU. 2009. C++: STL sort. (2009). https://gcc.gnu.org/onlinedocs/libstdc++/libstdc++-html-USERS-4.4/a01347.html

[17] Go. 2009. Source file src/sort/sort.go. (2009). https://golang.org/src/sort/sort.go

[18] Fuji Goro and Morwenn. 2019. Open-source C++ implementation of Timsort. (2019). https://github.com/gfx/cpp-TimSort

[19] Jim Gray, Chris Nyberg, Mehul Shah, and Naga Govindaraju. 2017. The SortBenchmark dataset. (2017). http://sortbenchmark.org/

[20] Chen-Yu Hsu, Piotr Indyk, Dina Katabi, and Ali Vakilian. 2019. Learning-Based Frequency Estimation Algorithms. In *International Conference on Learning Representations*. ICLR, New Orleans, LA. https://openreview.net/forum?id=r1lohoCqY7

[21] Hiroshi Inoue and Kenjiro Taura. 2015. SIMD-and cache-friendly algorithm for sorting an array of structures. *Proceedings of the VLDB Endowment* 8, 11 (2015), 1274–1285.

[22] Intel. 2020. Intel Performance Primitive library for x86 architectures. (2020). http://software.intel.com/en-us/intel-ipp/

[23] Java. 2017. Java 9: List.sort. (2017). https://docs.oracle.com/javase/9/docs/api/java/util/List.html#sort-java.util.Comparator-

[24] Nathan Jay, Noga H. Rotman, P. Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2018. Internet Congestion Control via Deep Reinforcement Learning. (2018). arXiv:cs.NI/1810.03259

[25] Johan Ludwig William Valdemar Jensen et al. 1906. Sur les fonctions convexes et les inégalités entre les valeurs moyennes. *Acta mathematica* 30 (1906), 175–193.

[26] Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R Nutter, and Jeremy D Schaub. 2017. Tencent Sort. (2017).

[27] Andreas Kipf, Thomas Kipf, Bernhard Radke, Viktor Leis, Peter Boncz, and Alfons Kemper. 2018. Learned Cardinalities: Estimating Correlated Joins with Deep Learning. (2018). arXiv:cs.DB/1809.00677

[28] Tim Kraska, Mohammad Alizadeh, Alex Beutel, Ed H. Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. SageDB: A Learned Database System. In *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research, Asilomar, CA, USA, January 13-16, 2019, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2019/papers/p117-kraska-cidr19.pdf

[29] Tim Kraska, Alex Beutel, Ed H Chi, Jeffrey Dean, and Neoklis Polyzotis. 2018. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*. ACM, 489–504.

[30] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[31] Maciej Kurant, Minas Gjoka, Carter T. Butts, and Athina Markopoulou. 2011. Walking on a Graph with a Magnifying Glass: Stratified Sampling via Weighted Random Walks. In *Proceedings of ACM SIGMETRICS '11*. San Jose, CA.

[32] Bohdan S Majewski, Nicholas C Wormald, George Havas, and Zbigniew J Czech. 1996. A family of perfect hashing methods. *Comput. J.* 39, 6 (1996), 547–554.

[33] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning Scheduling Algorithms for Data Processing Clusters. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, New York, NY, USA, 270–288. https://doi.org/10.1145/3341302.3342080

[34] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A Learned Query Optimizer. *Proc. VLDB Endow.* 12, 11 (July 2019), 1705–1718. https://doi.org/10.14778/3342263.3342644

[35] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. ACM, 3.

[36] Peter McIlroy. 1993. Optimistic sorting and information theoretic complexity. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 467–474.

[37] MongoDB. 2018. MongoDB: sorter.cpp. (2018). https://github.com/mongodb/mongo/blob/master/src/mongo/db/sorter/sorter.cpp

[38] David R Musser. 1997. Introspective sorting and selection algorithms. *Software: Practice and Experience* 27, 8 (1997), 983–993.

[39] MySQL. 2000. MySQL: filesort.cc. (2000). https://github.com/mysql/mysql-server/blob/8.0/sql/filesort.cc

[40] Omar Obeya, Endrias Kahssay, Edward Fan, and Julian Shun. 2019. Theoretically-Efficient and Practical Parallel In-Place Radix Sorting. In *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures*. ACM, 213–224.

[41] OpenAddresses. 2020. The OpenAddresses - Northeast dataset. (2020). https://data.openaddresses.io/openaddr-collected-us_northeast.zip

[42] OpenStreetMap contributors. 2017. Planet dump retrieved from https://planet.osm.org . (2017). https://www.openstreetmap.org

[43] Emanuel Parzen. 1962. On estimation of a probability density function and mode. *The annals of mathematical statistics* 33, 3 (1962), 1065–1076.

[44] Orson R. L. Peters. 2020. The Pattern-Defeating Quicksort Algorithm. (2020). https://github.com/orlp/pdqsort

[45] Tim Peters. 2002. Python: list.sort. (2002). https://github.com/python/cpython/blob/master/Objects/listsort.txt

[46] Postgres. 1996. Postgres: tuplesort.c. (1996). https://github.com/postgres/postgres/blob/master/src/backend/utils/sort/tuplesort.c

[47] Murray Rosenblatt. 1956. Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics* (1956), 832–837.

[48] Peter Sanders and Sebastian Winkel. 2004. Super scalar sample sort. In *European Symposium on Algorithms*. Springer, 784–796.

[49] Michael Axtmann Sascha Witt. 2020. Open-source C++ implementation of the IPS4o algorithm. (2020). https://github.com/SaschaWitt/ips4o

[50] Nadathur Satish, Changkyu Kim, Jatin Chhugani, Anthony D Nguyen, Victor W Lee, Daehyun Kim, and Pradeep Dubey. 2010. Fast sort on CPUs and GPUs: A case for bandwidth oblivious SIMD sort. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 351–362.

[51] Andrew Schein. 2009. Open-source C++ implementation of Radix Sort for double-precision floating points. (2009). https://bitbucket.org/ais/usort/src/474cc2a19224/usort/f8_sort.c

[52] SQLite. 2011. SQLite: vdbesort.c. (2011). https://github.com/mackyle/sqlite/blob/master/src/vdbesort.c

[53] Simon Steele and Marius Žilénas. 2020. 479k English words for all your dictionary. (2020). https://github.com/dwyl/english-words

[54] Michal Turčaník and Martin Javurek. 2016. Hash function generation by neural network. In *2016 New Trends in Signal Processing (NTSP)*. IEEE, 1–5.

[55] Peter Van Sandt, Yannis Chronis, and Jignesh M Patel. 2019. Efficiently Searching In-Memory Sorted Arrays: Revenge of the Interpolation Search?. In *Proceedings of the 2019 International Conference on Management of Data*. ACM, 36–53.

[56] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. 2014. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927* (2014).

[57] Jianfeng Wang, Jingdong Wang, Nenghai Yu, and Shipeng Li. 2013. Order preserving hashing for approximate nearest neighbor search. In *Proceedings of the 21st ACM international conference on Multimedia*. ACM, 133–142.

[58] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.