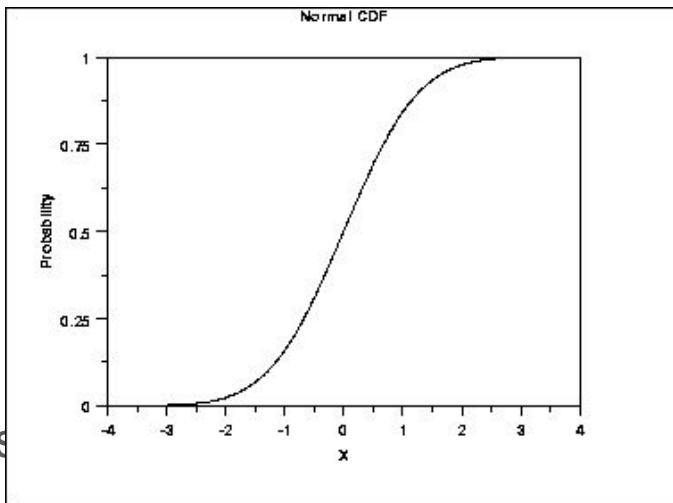


The Case for a Learned Sorting Algorithm

Ani Kirsto, Kapil Vaidya

Background: Sorting and CDF

- Sorting is a foundational problem in comp-sci
- Core Intuition: [CDF of sorted array]
 - CDF returns proportion of input below x
- In the case of the exact CDF with 0 duplicate values



$$pos = F_A(x) \cdot N = P(A \leq x) \cdot N.$$

Issues

- Unlikely to be able to create a perfect CDF of data
- Even if perfect CDF, maybe slower than Radix

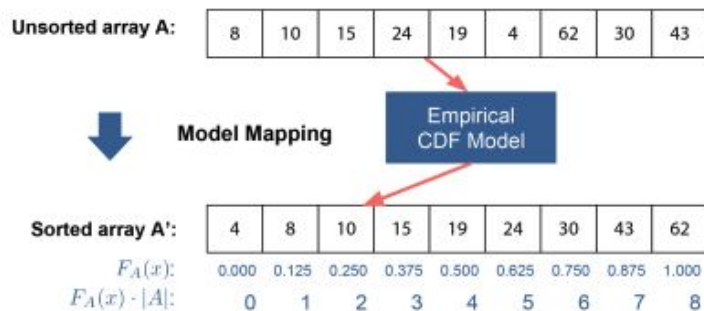


Figure 1: Sorting with the perfect CDF model

Imprecise Models, Drawbacks, Benefits

Collisions

- Imprecise model likely maps multiple keys into the same position in output array
- Spill Bucket: List of colliding elements => traditional sort & merge

Non-monotonic Model

- $a < b \not\Rightarrow F(a) < F(b)$ [output not perfectly sorted]
- Further work to fix imperfection [Insertion Sort]

Key Benefits

- Errors caused by non-monotonicity lead to small deviations in position in output
- Possibility to reduce collisions

Naive Algorithm

Collision-Handler = Spill Bucket

Param o : can be utilized to reduce collisions

- Compact over provisioned output array

Issues:

- Little cache locality
- Room for improvement in collision reduction

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  $A'$ 
```

Naive Algorithm Cache Locality Under Ideal Case

In the ideal case of array $A = [0 \dots 1e9]$, randomly permuted

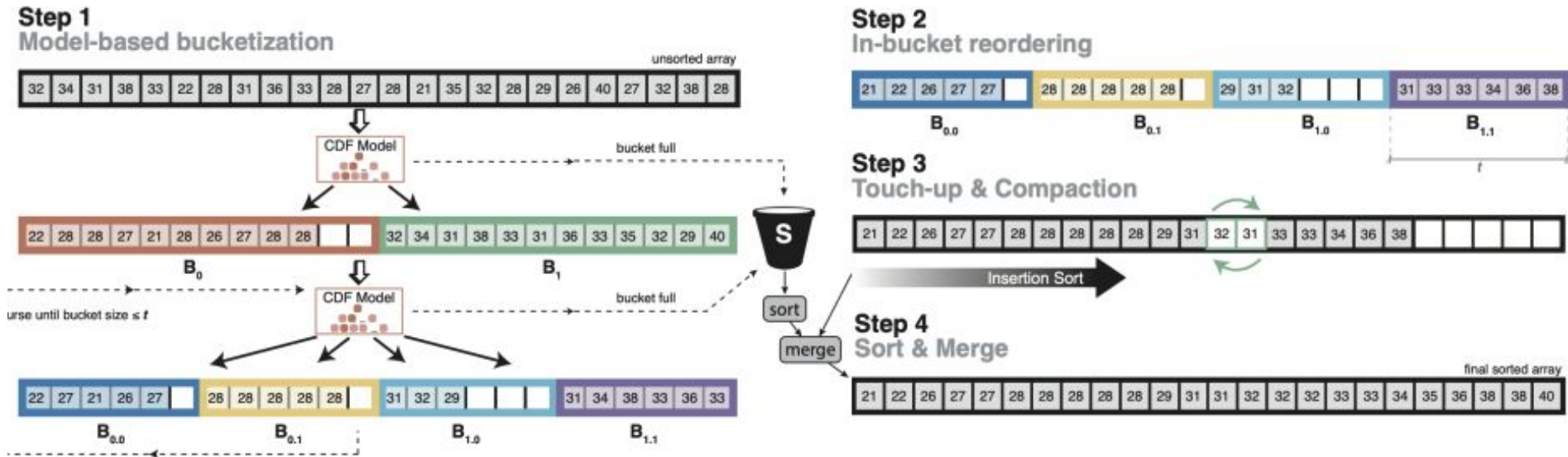
- Key = Index

Time to distribute keys in ideal case > cache optimized Radix Sort

- Inefficient => need cache local algorithm

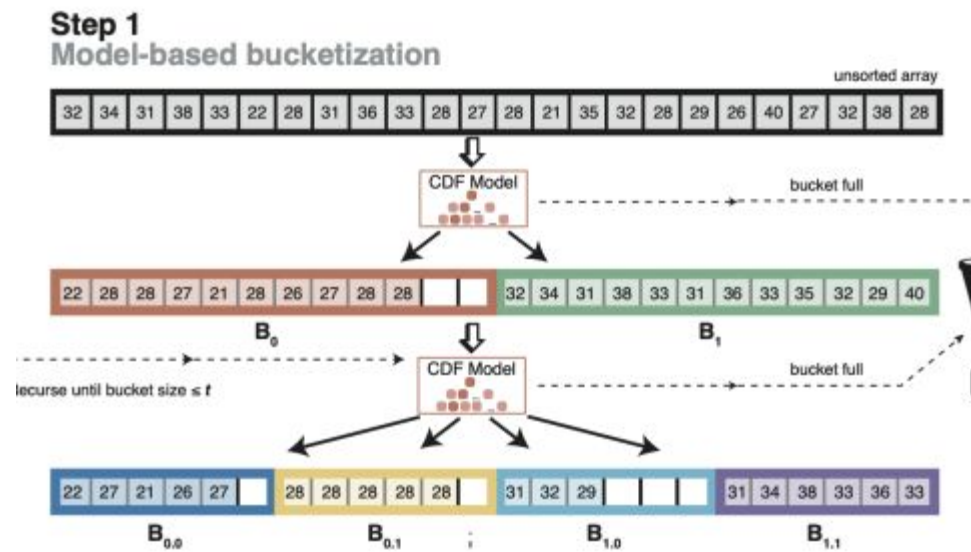
Algorithm 2 | Learned Sort

- Multi-level bucketing | param=[f, t]
- In Bucket Sorting [last-level]
- Compact and Insertion Sort
- Merge with Spill Buckets



Algorithm 2 Core - Bucketing

- Implementation Details:
 - Alternating Read and Write buckets [A & B]
- Fanout (f) parameter
 - Recursively split buckets into f B/f buckets
- Threshold (t)
 - Base case on bucket sizes
- Optimal Value f:
 - Large: Better use of model information
 - Small: Better cache locality [keep all buckets in cache]
- Optimal value t:
 - Large: Reduced Collisions
 - Small: More efficient in-bucket sort



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 ($E[s]/N$).

Alg 2: Bucketing

// Stage 1: Model-based bucketization

```
while  $b \geq t$  do                                ▶ Until bucket capacity reaches the threshold  $t$   
   $I \leftarrow [0] \times n$                                 ▶ Reset array  $I$   
  for  $x \in \text{read\_arr}$  do  
     $pos \leftarrow \lfloor \text{INFER}(F_A, x) \cdot n \rfloor$   
    if  $I[pos] \geq b$  then                                ▶ Bucket is full  
       $S.\text{append}(x)$                                 ▶ Add to spill bucket  
    else                                ▶ Write into the predicted bucket  
       $\text{write\_arr}[pos \cdot b + I[pos]] \leftarrow x$   
      INCREMENT  $I[pos]$   
   $b \leftarrow \lfloor b/f \rfloor$                                 ▶ Update bucket capacity  
   $n \leftarrow \lfloor N/b \rfloor$                                 ▶ Update the number of buckets  
  PTRSWP(read_arr, write_arr)                        ▶ Pointer swap to reuse memory
```

Algorithm 2 - In Bucket Reordering

- Count number of items mapping to diff bucket indices
- Prefix Sum [& self]
 - Use as index offset to insert, decrement for next position

// Stage 2: In-bucket reordering

$\text{offset} \leftarrow 0$

for $i \leftarrow 0$ **up to** n **do**

$K \leftarrow [0] \times b$

‣ Process each bucket

‣ Array of counts

for $j \leftarrow 0$ **up to** $I[i]$ **do** ‣ Record the counts of the predicted positions

$\text{pos} \leftarrow \lfloor \text{INFER}(F_A, \text{read_arr}[\text{offset} + j]) \cdot N \rfloor$

INCREMENT $K[\text{pos} - \text{offset}]$

for $j \leftarrow 1$ **up to** $|K|$ **do**

$K[j] \leftarrow K[j] + K[j - 1]$

‣ Calculate the running total

$T \leftarrow []$

‣ Temporary auxiliary memory

for $j \leftarrow 0$ **up to** $I[i]$ **do**

‣ Order keys w.r.t. the cumulative counts

$\text{pos} \leftarrow \lfloor \text{INFER}(F_A, \text{read_arr}[\text{offset} + j]) \cdot N \rfloor$

$T[j] \leftarrow \text{read_arr}[\text{offset} + K[\text{pos} - \text{offset}]]$

DECREMENT $K[\text{pos} - \text{offset}]$

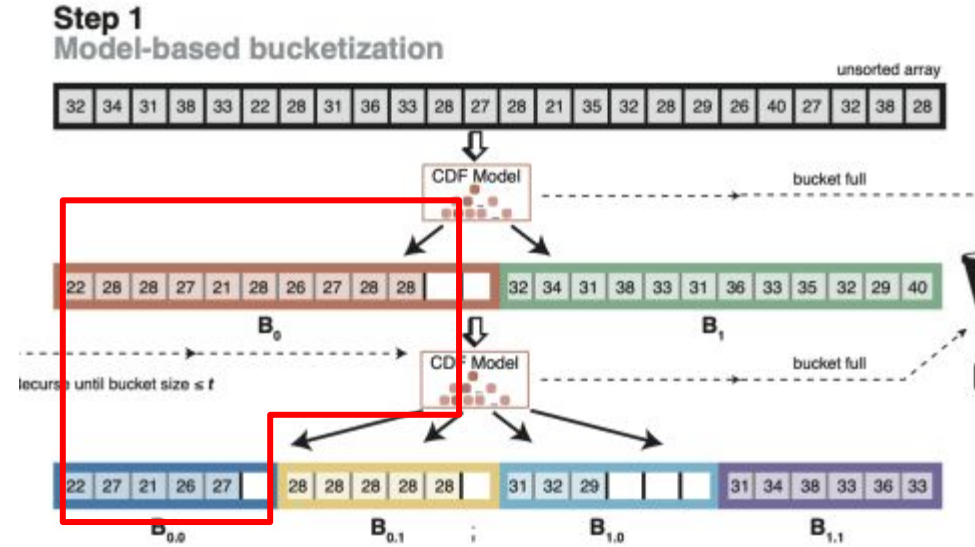
Copy T back to $\text{read_arr}[\text{offset}]$

$\text{offset} \leftarrow \text{offset} + b$

Implemented Optimizations

- Use model in batches [cache optimal]
- Overprovision arrays [1.1x] => reduce collisions
- One bucket at a time
 - Recurse on one bucket at a time
 - Cache optimal

```
for  $x \in \text{read\_arr}$  do  
   $\text{pos} \leftarrow \lfloor \text{INFER}(F_A, x) \cdot n \rfloor$ 
```



CDF Model: RMI

- RMI=Recursive Model Index
- Cheap to train, quick inference
- Structure
 - Multi-Layer fanout

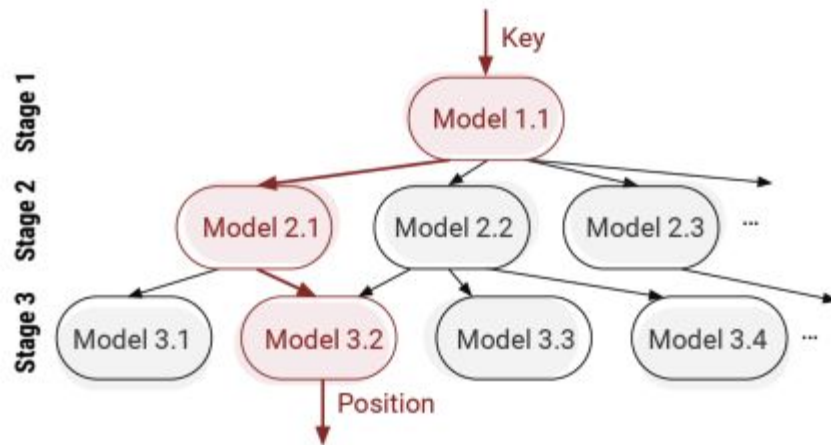


Figure 5: A typical RMI architecture containing three layers

- Deeper layers specialize [learn output range at fine granularity]
- Selected to use Linear Spline Fitting
 - Better Monotonicity
 - Faster training

RMI Training

- Sample Data
- Train Layer
- Scale output to be able to select model in next layer
- Pass subset of data to next layer

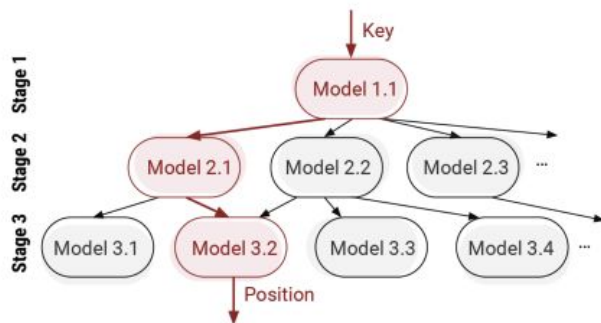


Figure 5: A typical RMI architecture containing three layers

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:    $S \leftarrow \text{SAMPLE}(A)$ 
3:   SORT( $S$ )
4:    $T \leftarrow [][][]$  ▷ Training sets implemented as a 3D array
5:   for  $i \leftarrow 0$  up to  $|S|$  do
6:      $T[0][0].\text{add}((S[i], i/|S|))$ 
7:   for  $l \leftarrow 0$  up to  $L$  do
8:     for  $m \leftarrow 0$  up to  $M^l$  do
9:        $F_A[l][m] \leftarrow \text{linear model trained on the set } \{t \mid t \in T[l][m]\}$ 
10:      if  $l + 1 < L$  then
11:        for  $t \in T[l][m]$  do
12:           $F_A[l][m].\text{slope} \leftarrow F_A[l][m].\text{slope} \cdot M^{l+1}$ 
13:           $F_A[l][m].\text{intercept} \leftarrow F_A[l][m].\text{intercept} \cdot M^{l+1}$ 
14:           $i \leftarrow F_A[l][m].\text{slope} \cdot t + F_A[l][m].\text{intercept}$ 
15:           $T[l + 1][i].\text{add}(t)$ 
16:   return  $F_A$ 

```

RMI Inference

- Recursively select model
 - Feed same data back into each model selected

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:      $r = x \cdot F_A[l][r].\text{slope} + F_A[l][r].\text{intercept}$ 
7:   return  $r$ 
```

Analysis

- $E[\text{in spill bucket}]$ when $t = 1$
 - Analogous to random hashing

Sample Size

- Dvoretzky-Kiefer-Wolfowitz Inequality $\Rightarrow O(1/\sqrt{n})$ difference between sample and real CDF
- Large Sample Size $\Rightarrow$ Large training time

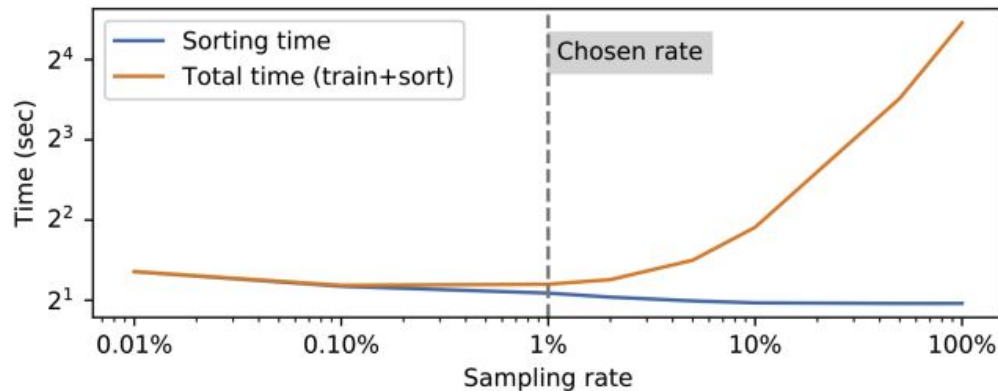


Figure 7: Sorting time for various sampling rates of 100M normally-distributed keys (log-log scale).

Complexity

- N = number items
 - L = # layers in RMI Model,
 - t = bucket recursion basecase
 - s = size of spill bucket
-
- Stage 1: Approximately $O(N \cdot L)$ cost of calling model
 - Stage 2: In Bucket Sort $O(n)$
 - Stage 3: Compaction & Insertion Sort = $O(Nt)$
 - Stage 4: $O(s \log s) + O(n)$

Sorting Strings

- Monotonically encode characters
- Values [0..1] & Monotonic but not good CDF
- Normalize => [outputs will be clustered => need to learn a normalization]
- Model learns roughly equally sized buckets

$$\frac{C_1}{256} + \frac{C_2}{256^2} + \dots + \frac{C_n}{256^n}.$$

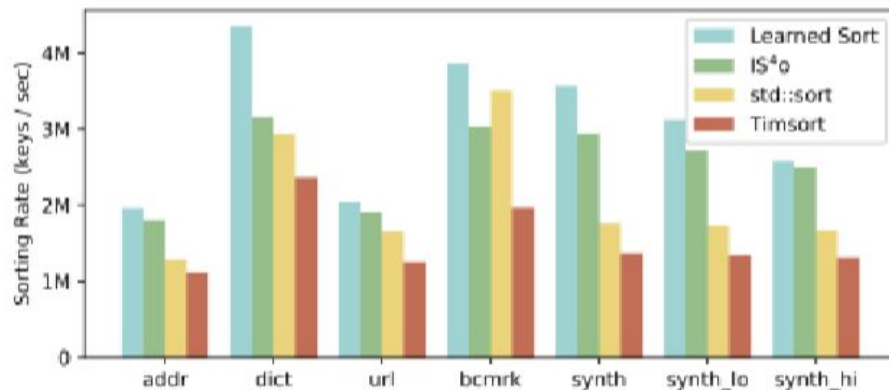


Figure 10: The sorting rate for various strings datasets.

Evaluation

- **Model training time included as part of overall time**
- Comparison against
 - C++ Radix Sort
 - Timsort
 - IS⁴o sort
- Measured by Sorting Frequency (keys sorted / sec), higher = better
- Light Teal = Learned Sort

Overall Performance

- Average 30% higher throughput than next best
- 55% compared to radix
- Comparable to Radix when data fits in L3 cache

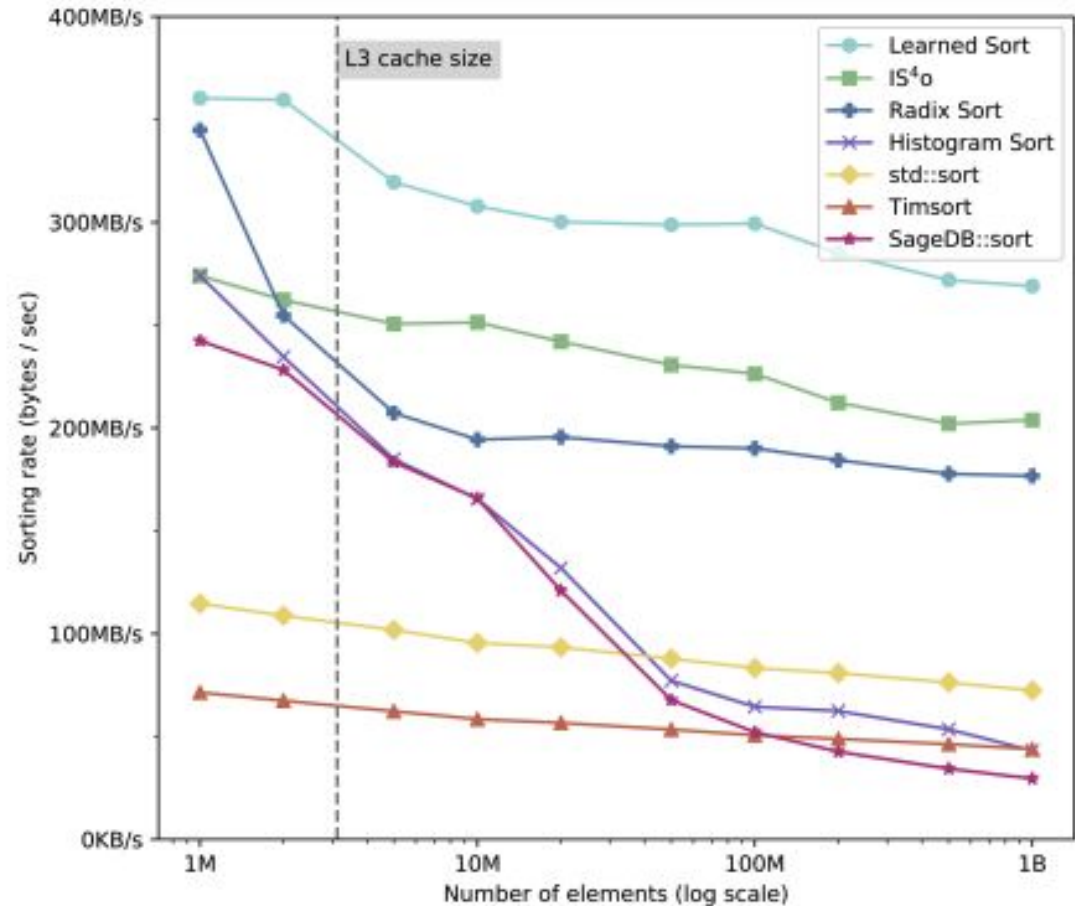


Figure 8: The sorting throughput for normally distributed double-precision keys (higher is better).

Across Datasets

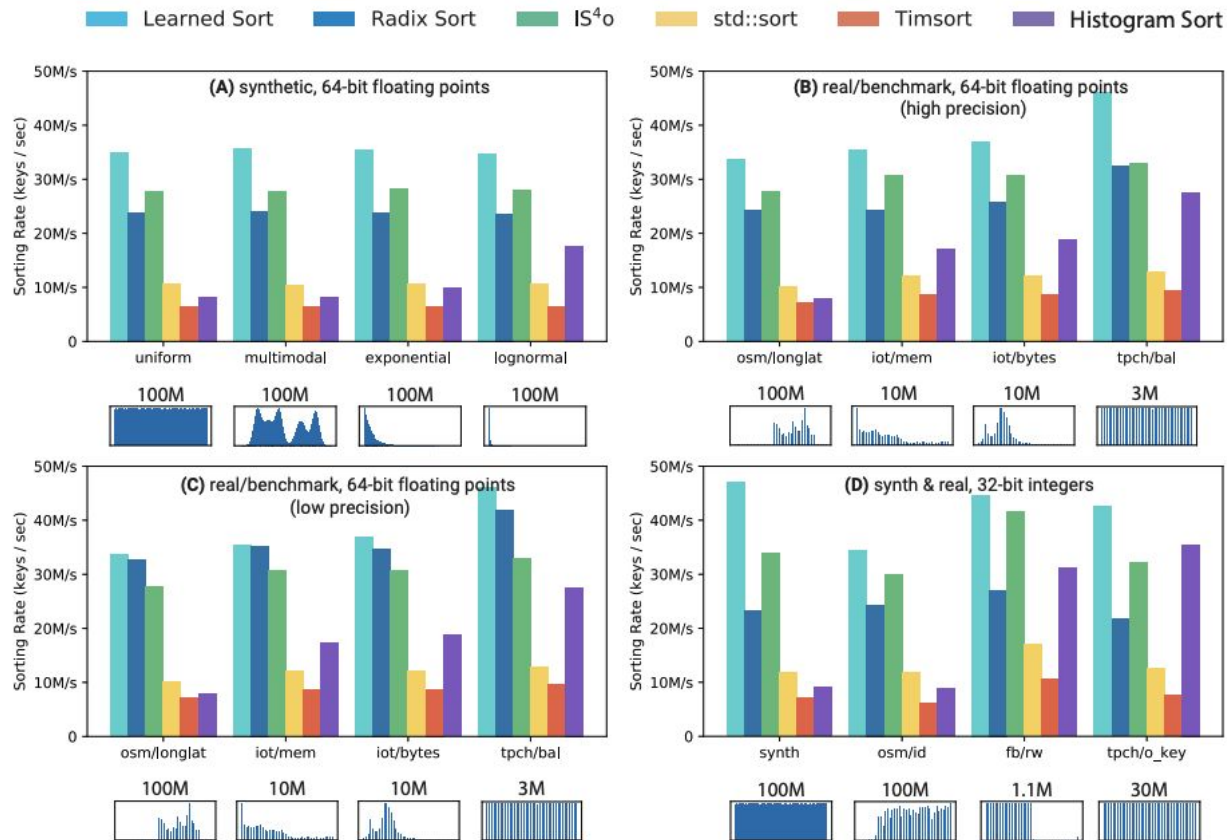


Figure 9: The sorting rate of Learned Sort and other baselines for real and synthetic datasets containing both doubles and integers. The pictures below the charts visualize the key distributions and the dataset sizes.

Model Choice

- Previous choice of model effective.

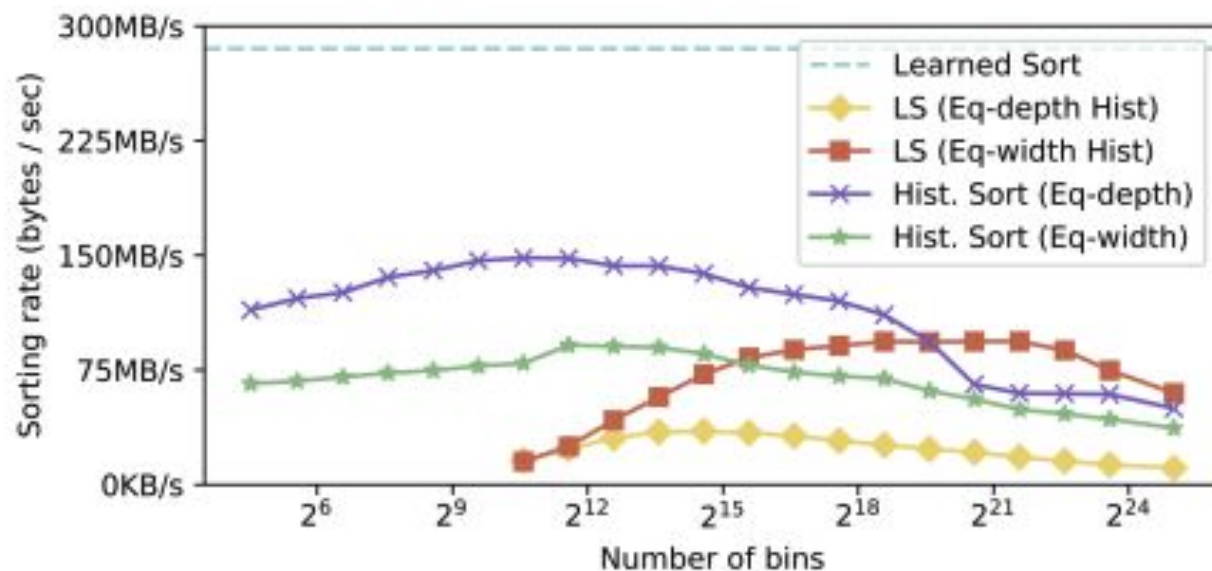


Figure 14: The sorting rate of Learned Sort algorithm on 100M normally-distributed keys as compared with (1) a version of LS that uses an equi-depth histogram as CDF model, (2) a version with an equi-width histogram, (3) Equi-depth Histogram Sort, and (4) Equi-width Histogram Sort.

Per Stage Performance



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

Conclusion

- Novel approach to accelerate sorting using a learned approximate of the CDF of the distribution
- Significant performance improvements compared to existing competitive algorithms.
- Future Work: Handling Strings, Complex Types, Parallel Sorting

Takeaways

- Strengths:
 - Integrated hardware optimization with general approach to increase performance.
 - Effectively explains trade offs in algorithm parameter selection.
- Weaknesses:
 - Lack of comparison against “optimal” 0-overhead case when implemented w/ cache improved bucketization
 - Non-rigorous runtime analysis
 - No heuristic for selecting model parameters fan-out (f) & threshold (t)
- Questions
 - Parallelization of algorithm
 - Exact implementation of choice of model
 - Difference between ideal & ideal w/ Bucketization