The Case for a Learned Sorting Algorithm

Authors: Ani Kristo, Kapil Vaidya, Ugur Cetintemel, Sanchit Misra, Tim Kraska

Presenter: Terryn Brunelle

Motivation

- Fundamental CS problem
- Database operations
  - Sort query results
  - Perform joins

Existing Work

- Comparison sort
- Distribution sort
  - Counting sort
  - Radix sort
- ML-enhanced algorithms

Learned Sort

- Train CDF model
- Use predicted prob for each key to predict final position for every key in sorted output

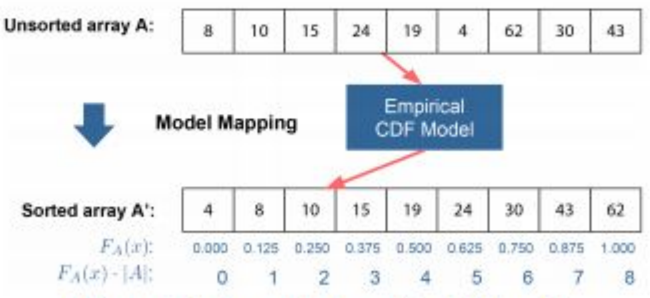


Figure 1: Sorting with the perfect CDF model

Linear time possible!
(if have perfect model)

Problems

- Perfect model = expensive to train
- Random-access memory problem

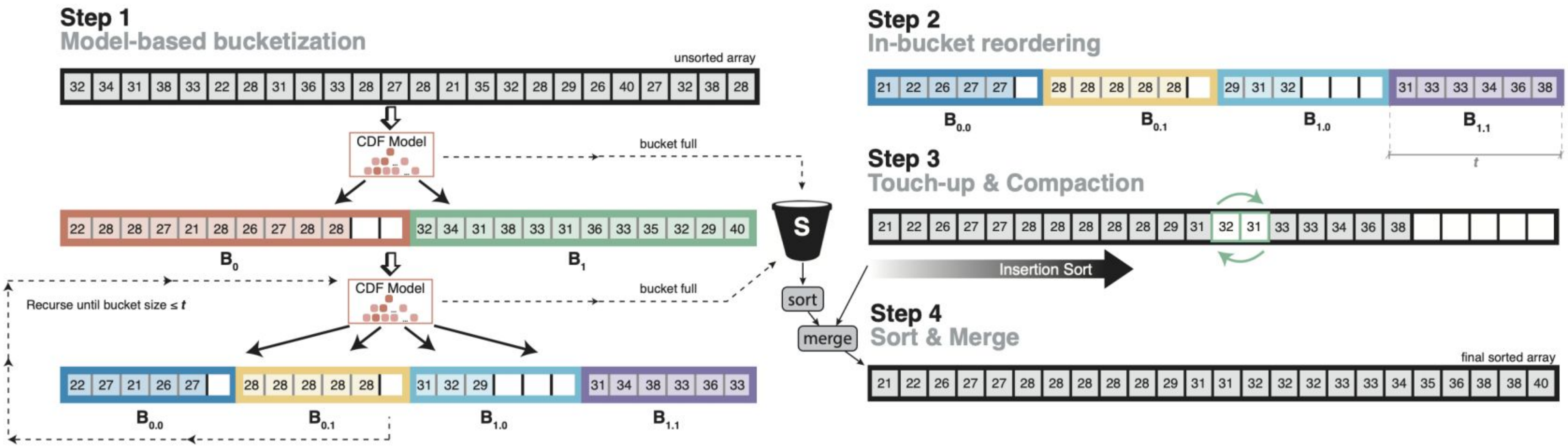
Algorithm 1

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

procedure LEARNED-SORT(A, F_A, o)
    $N \leftarrow A$.length
    $A' \leftarrow$ empty array of size $(N \cdot o)$
    for x **in** A **do**
        pos $\leftarrow \lfloor F_A(x) \cdot N \cdot o \rfloor$
        if EMPTY($A'[pos]$) **then** $A'[pos] \leftarrow x$
        else COLLISION-HANDLER(x)
    if $o > 1$ **then** COMPACT(A')
    if NON-MONOTONIC **then** INSERTION-SORT(A')
    return A'

Cache-Efficient Learned Sort



Pseudo-Code: Step 1

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

```
procedure LEARNED-SORT(A, F_A, f, t)
    N ← |A|                                  ▹ Size of the input array
    n ← f                                    ▹ n represents the number of buckets
    b ← ⌊N/f⌋                                ▹ b represents the bucket capacity
    B ← [] × N                               ▹ Empty array of size N
    I ← [0] × n                              ▹ Records bucket sizes
    S ← []                                   ▹ Spill bucket
    read_arr ← pointer to A
    write_arr ← pointer to B

    // Stage 1: Model-based bucketization
    while b ≥ t do                           ▹ Until bucket capacity reaches the threshold t
        I ← [0] × n                          ▹ Reset array I
        for x ∈ read_arr do
            pos ← ⌊INFER(F_A, x) · n⌋
            if I[pos] ≥ b then               ▹ Bucket is full
                S.append(x)                  ▹ Add to spill bucket
            else                             ▹ Write into the predicted bucket
                write_arr[pos · b + I[pos]] ← x
                INCREMENT I[POS]
        b ← ⌊b/f⌋                            ▹ Update bucket capacity
        n ← ⌊N/b⌋                            ▹ Update the number of buckets
        PTRSWP(read_arr, write_arr)          ▹ Pointer swap to reuse memory
```

Pseudo-Code: Steps 2-4

```
// Stage 2: In-bucket reordering
offset ← 0
for i ← 0 up to n do                                  ▹ Process each bucket
    K ← [0] × b                                       ▹ Array of counts

    for j ← 0 up to I[i] do    ▹ Record the counts of the predicted positions
        pos ← ⌊INFER(F_A, read_arr[offset + j]) · N⌋
        INCREMENT K[pos − offset]

    for j ← 1 up to |K| do                            ▹ Calculate the running total
        K[j] ← K[j] + K[j − 1]

    T ← []                                            ▹ Temporary auxiliary memory
    for j ← 0 up to I[i] do          ▹ Order keys w.r.t. the cumulative counts
        pos ← ⌊INFER(F_A, read_arr[offset + j]) · N⌋
        T[j] ← read_arr[offset + K[pos − offset]]
        DECREMENT K[pos − offset]
    Copy T back to read_arr[offset]
    offset ← offset + b

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

// Stage 4: Sort & Merge
SORT(S)
A′ ← MERGE(read_arr, S)

return A′
```

Optimizations

- Process elements in batches (cache locality)
- One bucket at a time (temporal locality)
- Bucket buffer space (reduce overflows)

CDF Model

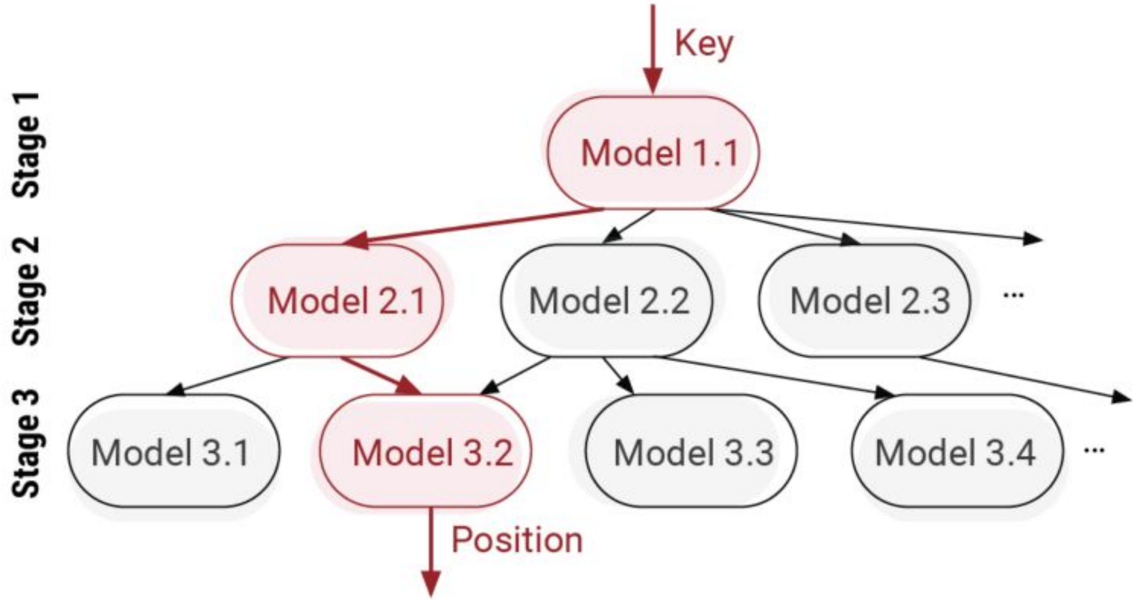


Figure 5: A typical RMI architecture containing three layers

CDF Model

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)

```
procedure INFER(F_A, x)
    L ← the number of layers of the CDF model F_A
    M^l ← the number of models in the l^th layer of the RMI F_A
    r ← 0
    for l ← 0 up to L do
        r = x · F_A[l][r].slope + F_A[l][r].intercept
    return r
```

Theoretical Results

- Step 1: O(N*L)
- Step 2: O(N)
- Step 3: O(Nt) (non-dominant)
- Step 4: O(s log s) + O(N)

Space complexity: order of O(N)

Experimental Results

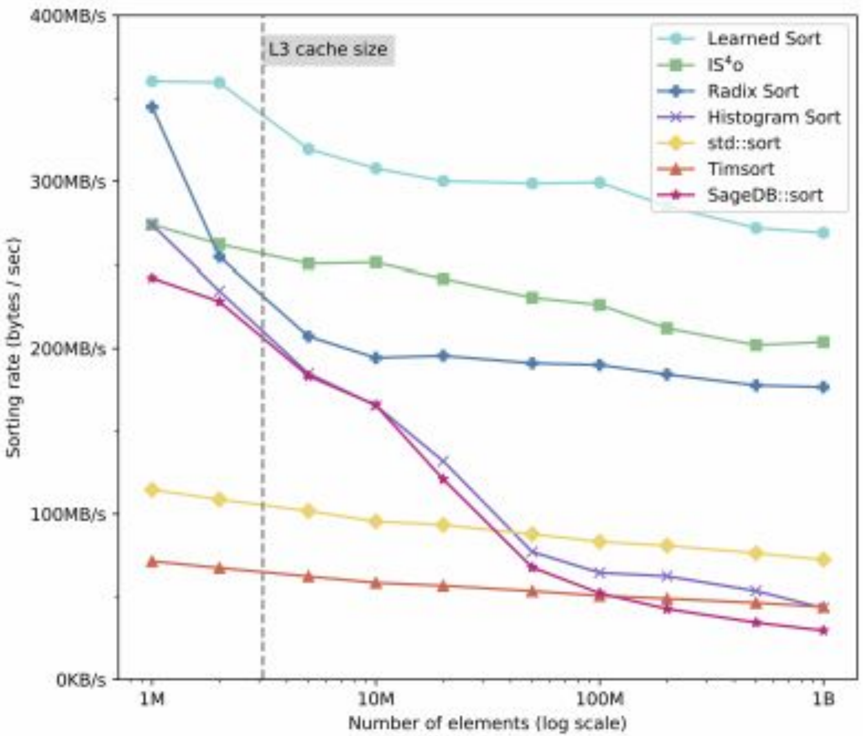
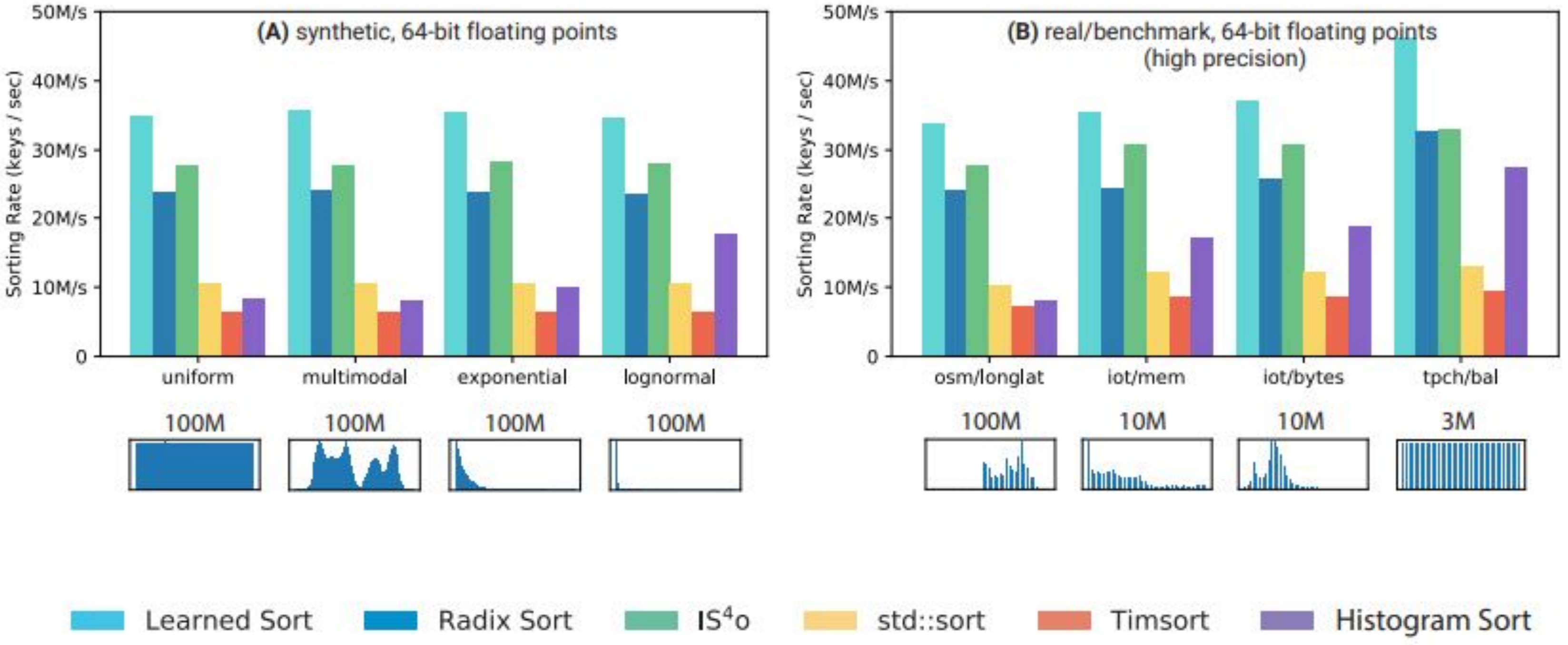
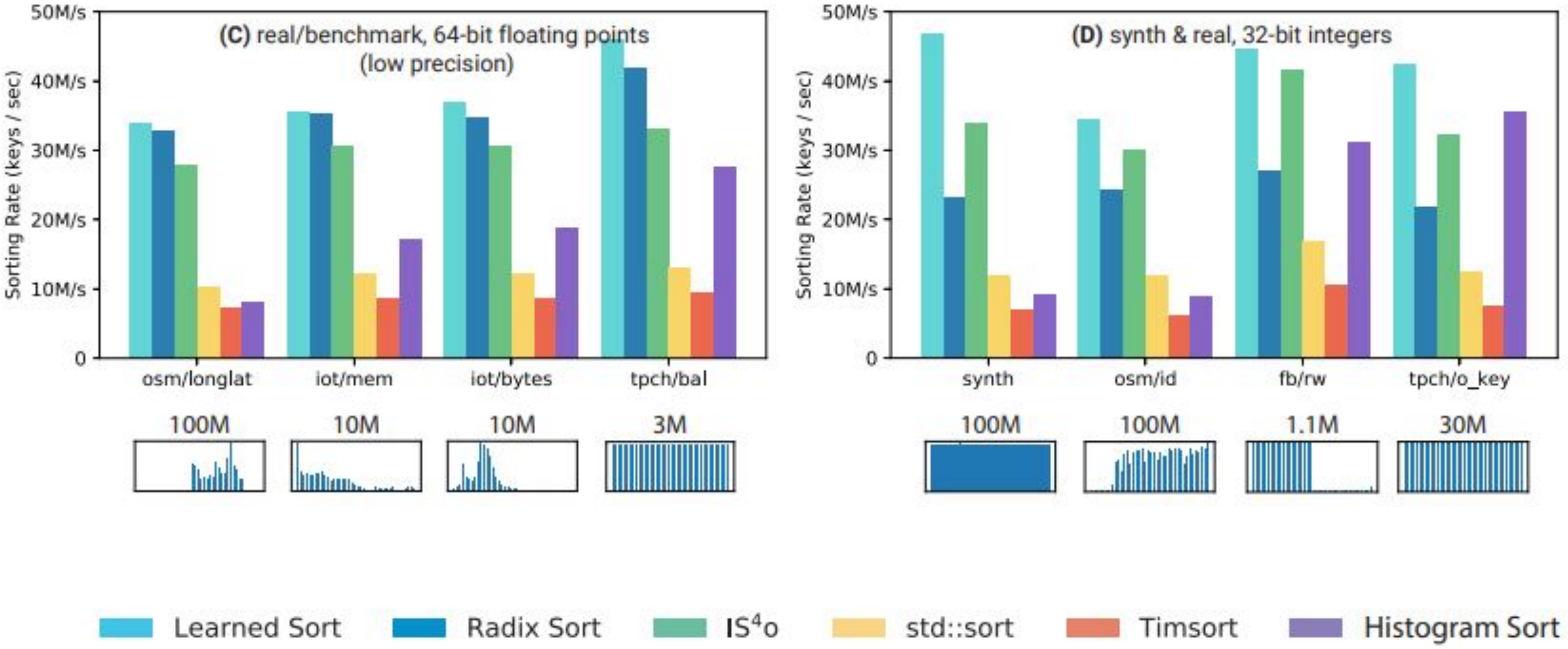


Figure 8: The sorting throughput for normally distributed double-precision keys (higher is better).

Experimental Results



Experimental Results



In-Place Sorting

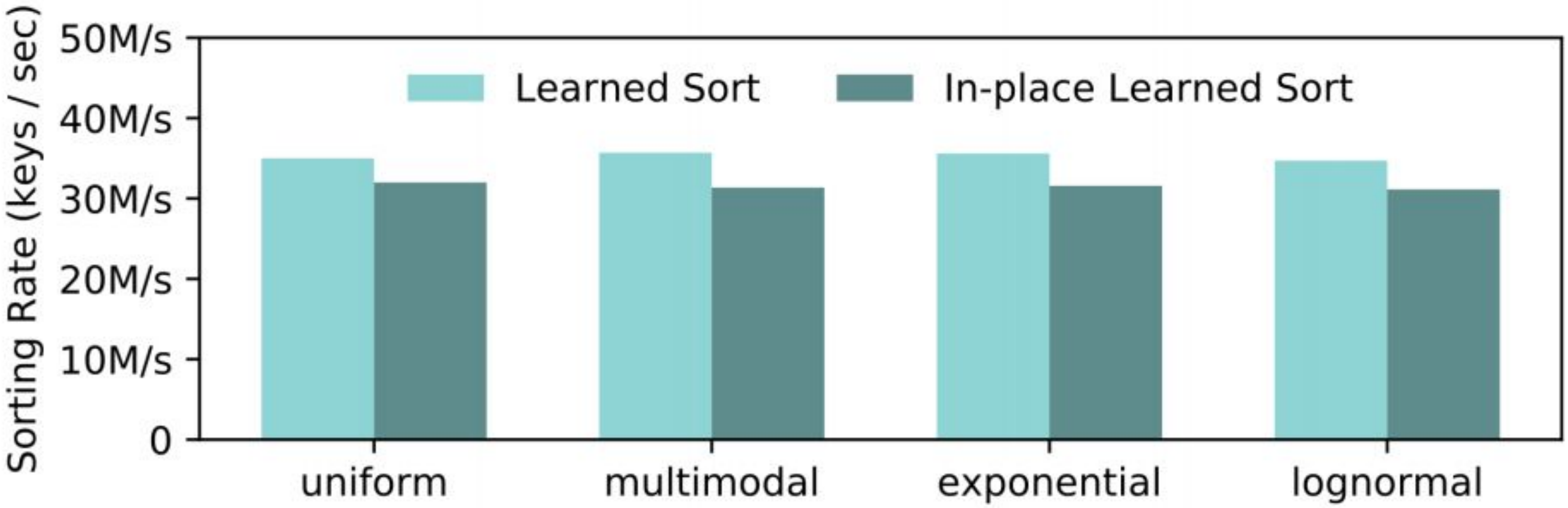


Figure 12: The sorting rate of Learned Sort and its in-place version for all of our synthetic datasets.

Performance Decomposition

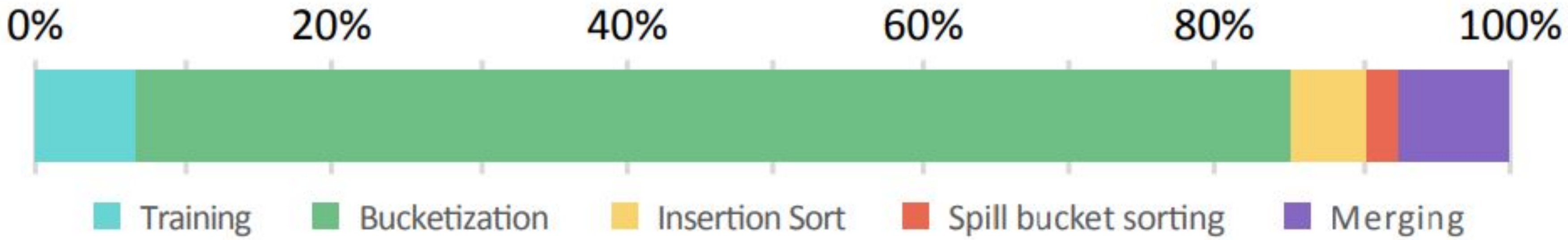


Figure 13: Performance of each of the stages of Learned Sort.

Strengths/Weaknesses

Strengths

- Performance on real-world data
- Improvement over default Java/Python sorting function
- Cache-efficient
- Model training time accounted for

Weaknesses

- Other CDF implementations?
- Duplicate keys

Directions for Future Work

- Sorting complex objects
- Parallel Sorting
- Using in DB systems

Discussion Questions

- Can you think of adversarial inputs that may be good to evaluate this specific approach on?
- What parallelization techniques may apply to this algorithm/sorting algorithms in general?
- What are other ways through which collisions might be handled? What is attractive about the spill bucket method?

Additional Materials

String Sorting

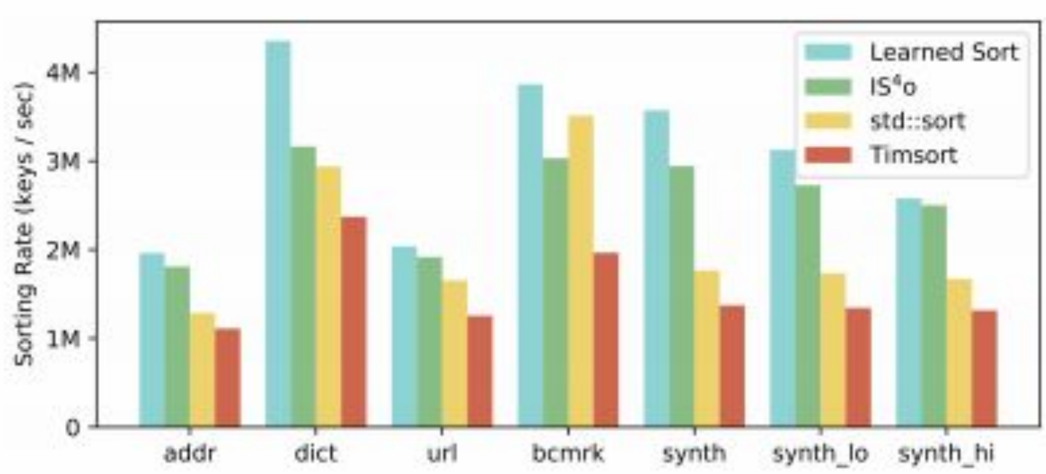
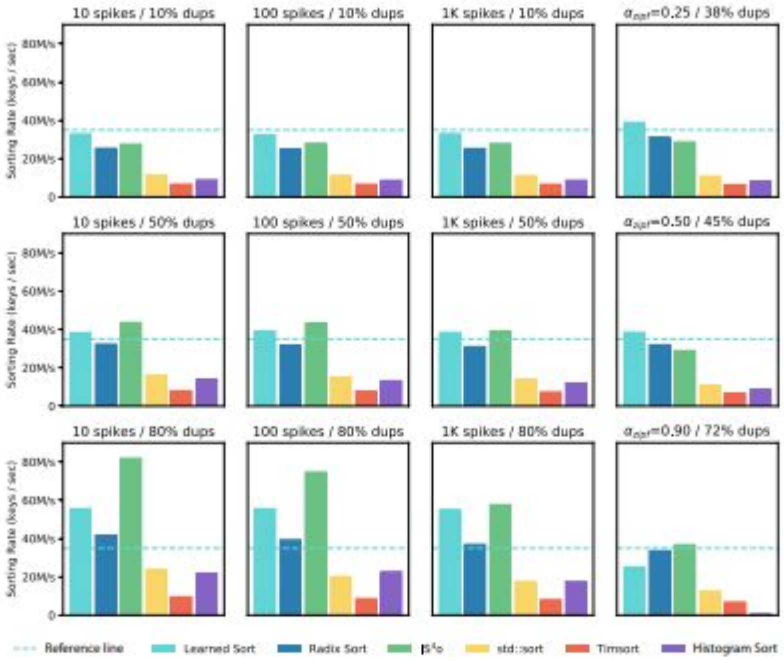


Figure 10: The sorting rate for various strings datasets.

Duplicates



10 spikes / 10% dups
100 spikes / 10% dups
1K spikes / 10% dups
α_{zipf}=0.25 / 38% dups
10 spikes / 50% dups
100 spikes / 50% dups
1K spikes / 50% dups
α_{zipf}=0.50 / 45% dups
10 spikes / 80% dups
100 spikes / 80% dups
1K spikes / 80% dups
α_{zipf}=0.90 / 72% dups
Sorting Rate (keys / sec)
80M/s
60M/s
40M/s
20M/s
0
Reference line
Learned Sort
Radix Sort
IS⁴o
std::sort
Timsort
Histogram Sort

CDF Model Training

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

procedure TRAIN(A, L, M)
    $S \leftarrow$ SAMPLE(A)
    SORT(S)
    $T \leftarrow [][][]$ ▷ Training sets implemented as a 3D array
    for $i \leftarrow 0$ **up to** $|S|$ **do**
        $T[0][0]$.add(($S[i]$, $i/|S|$))
    for $l \leftarrow 0$ **up to** L **do**
        for $m \leftarrow 0$ **up to** M^l **do**
            $F_A[l][m] \leftarrow$linear model trained on the set $\{t \mid t \in T[l][m]\}$
            if $l + 1 < L$ **then**
                for $t \in T[l][m]$ **do**
                    $F_A[l][m]$.slope$\leftarrow F_A[l][m]$.slope $\cdot M^{l+1}$
                    $F_A[l][m]$.intercept$\leftarrow F_A[l][m]$.intercept $\cdot M^{l+1}$
                    $i \leftarrow F_A[l][m]$.slope $\cdot t + F_A[l][m]$.intercept
                    $T[l+1][i]$.add(t)
    return F_A