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

Linear time possible! (if have perfect model)
Problems

- Perfect model = expensive to train
- Random-access memory problem
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 $A'$
Cache-Efficient Learned Sort

**Step 1**
Model-based bucketization

**Step 2**
In-bucket reordering

**Step 3**
Touch-up & Compaction

**Step 4**
Sort & Merge

!!!Note: The diagram illustrates the process of converting an unsorted array into a sorted array using a cache-efficient learned sort method. The steps include model-based bucketization, in-bucket reordering, touch-up & compaction, and sort & merge. Each step is visually represented with intermediate sorted buckets and final sorted array.!!!
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 $A$

1: procedure Learned-Sort($A$, $F_A$, $f$, $t$)
2: \[ N \leftarrow |A| \] \hspace{1em} \triangleright \text{Size of the input array}
3: \[ n \leftarrow f \] \hspace{1em} \triangleright \text{$n$ represents the number of buckets}
4: \[ b \leftarrow \lfloor N/f \rfloor \] \hspace{1em} \triangleright \text{$b$ represents the bucket capacity}
5: \[ B \leftarrow [] \times N \] \hspace{1em} \triangleright \text{Empty array of size $N$}
6: \[ I \leftarrow [0] \times n \] \hspace{1em} \triangleright \text{Records bucket sizes}
7: \[ S \leftarrow [] \] \hspace{1em} \triangleright \text{Spill bucket}
8: \[ \text{read}_\text{arr} \leftarrow \text{pointer to } A \]
9: \[ \text{write}_\text{arr} \leftarrow \text{pointer to } B \]

10: // Stage 1: Model-based bucketization
11: while $b \geq t$ do \hspace{1em} \triangleright \text{Until bucket capacity reaches the threshold $t$}
12: \[ I \leftarrow [0] \times n \] \hspace{1em} \triangleright \text{Reset array $I$}
13: \[ \text{for } x \in \text{read}_\text{arr} \text{ do} \]
14: \[ \text{pos} \leftarrow \lfloor \text{Infer}(F_A, x) \cdot n \rfloor \]
15: \[ \text{if $I[\text{pos}] \geq b$ then} \] \hspace{1em} \triangleright \text{Bucket is full}
16: \[ S.\text{append}(x) \] \hspace{1em} \triangleright \text{Add to spill bucket}
17: \[ \text{else} \] \hspace{1em} \triangleright \text{Write into the predicted bucket}
18: \[ \text{write}_\text{arr}[\text{pos} \cdot b + I[\text{pos}]] \leftarrow x \]
19: \[ \text{INCREMENT } I[\text{pos}] \]
20: \[ b \leftarrow \lfloor b/f \rfloor \] \hspace{1em} \triangleright \text{Update bucket capacity}
21: \[ n \leftarrow \lfloor N/b \rfloor \] \hspace{1em} \triangleright \text{Update the number of buckets}
22: \[ \text{PtrSwap(read}_\text{arr}, \text{write}_\text{arr}) \] \hspace{1em} \triangleright \text{Pointer swap to reuse memory}
Pseudo-Code: Steps 2-4

23:     // Stage 2: In-bucket reordering
24:     offset ← 0
25:     for i ← 0 up to n do                       ➔ Process each bucket
26:         K ← [0] × b                          ➔ Array of counts
27:             for j ← 0 up to I[i] do          ➔ Record the counts of the predicted positions
28:                 pos ← ⌈INFERR(A, read_arr[offset + j]) ⋅ N⌉
29:                 INCREMENT K[pos − offset]
30:             for j ← 1 up to |K| do          ➔ Calculate the running total
32:             T ← []                           ➔ Temporary auxiliary memory
33:             for j ← 0 up to I[i] do          ➔ Order keys w.r.t. the cumulative counts
34:                 pos ← ⌈INFERR(A, read_arr[offset + j]) ⋅ N⌉
35:                 T[j] ← read_arr[offset + K[pos − offset]]
36:                 DECREMENT K[pos − 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′ ← MERGE(read_arr, S)
44:     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
**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: \quad L \leftarrow \text{the number of layers of the CDF model } F_A
3: \quad M^l \leftarrow \text{the number of models in the } l^{th} \text{ layer of the RMI } F_A
4: \quad r \leftarrow 0
5: \quad \textbf{for } l \leftarrow 0 \textbf{ up to } L \textbf{ do}
6: \quad \quad r = x \cdot F_A[l][r].\text{slope} + F_A[l][r].\text{intercept}
7: \quad \textbf{return } r
Theoretical Results

- Step 1: $O(N \times L)$
- Step 2: $O(N)$
- Step 3: $O(N_t)$ (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

(A) synthetic, 64-bit floating points

(B) real/benchmark, 64-bit floating points (high precision)

Sorting Rate (keys/sec)

- uniform
- multimodal
- exponential
- lognormal

100M

100M

100M

100M

100M

10M

10M

10M

3M

Learned Sort
Radix Sort
IS^4o
std::sort
Timsort
Histogram Sort
Experimental Results

(C) real/benchmark, 64-bit floating points (low precision)

(D) synth & real, 32-bit integers

Sorting Rate (keys/sec)

Learned Sort  Radix Sort  IS^4o  std::sort  Timsort  Histogram Sort
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
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

1: procedure $\text{Train}(A, L, M)$  
2: $S \leftarrow \text{Sample}(A)$  
3: Sort($S$)  
4: $T \leftarrow [||]$  
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: $F_A[l][m] \leftarrow \text{linear model trained on the set } \{ t : t \in T[l][m] \}$  
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 $F_A$