Machine Learning for Data Management Systems

Sorting; Joins

Amol Deshpande
February 28, 2023

Motivation

- Sorting and joins are key operations in data management systems

- Can we make them faster through use of CDFs?

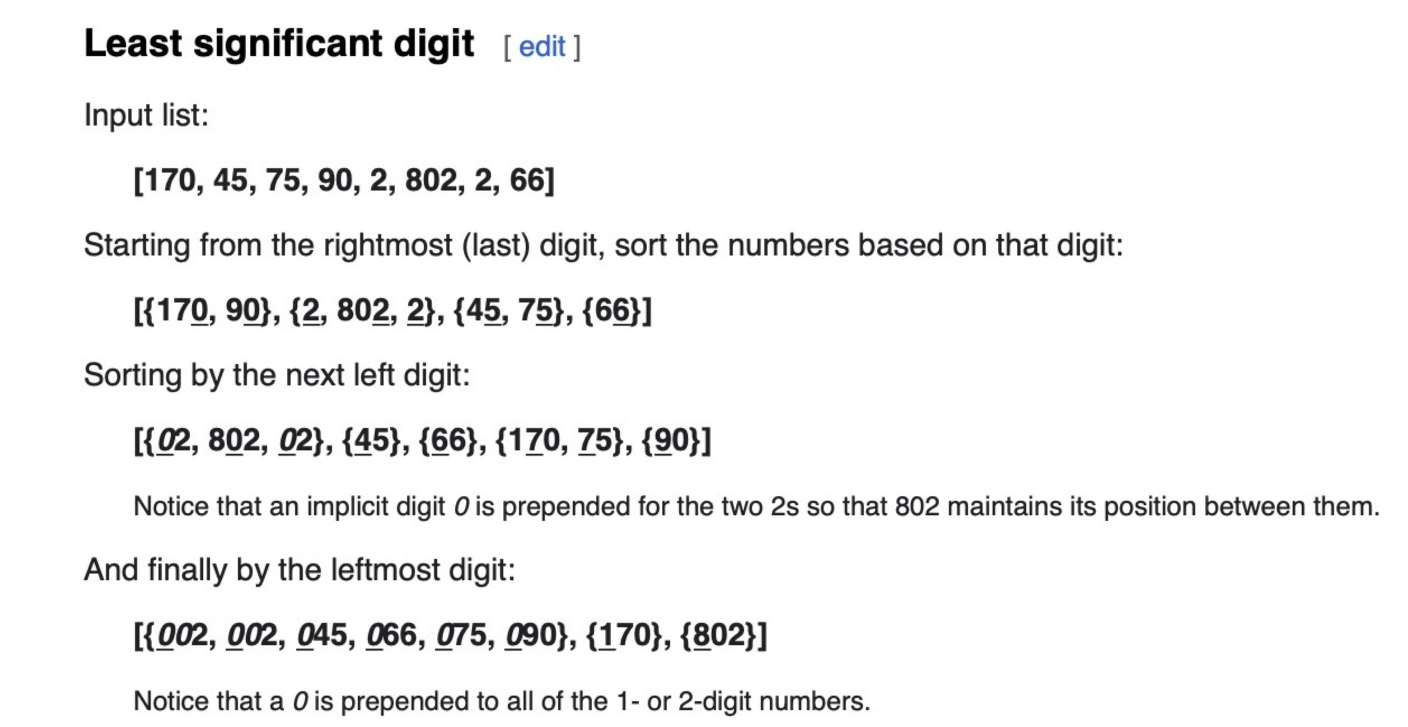
- Distribution information has been used for many years for optimizing these operations
  - E.g., to deal with skew, to decide how to partition data, etc.

Outline

- Sorting
- Joins

Sorting

- Simplest algorithm: Quicksort
- RadixSort



Least significant digit

Input list:

[170, 45, 75, 90, 2, 802, 2, 66]

Starting from the rightmost (last) digit, sort the numbers based on that digit:

[{170, 90}, {2, 802, 2}, {45, 75}, {66}]

Sorting by the next left digit:

[{*0*2, 802, *0*2}, {45}, {66}, {170, 75}, {90}]

Notice that an implicit digit *0* is prepended for the two 2s so that 802 maintains its position between them.

And finally by the leftmost digit:

[{*00*2, *00*2, *0*45, *0*66, *0*75, *0*90}, {170}, {802}]

Notice that a *0* is prepended to all of the 1- or 2-digit numbers.

Sorting with Perfect CDF

- Scan the relation, and copy each item to the right location

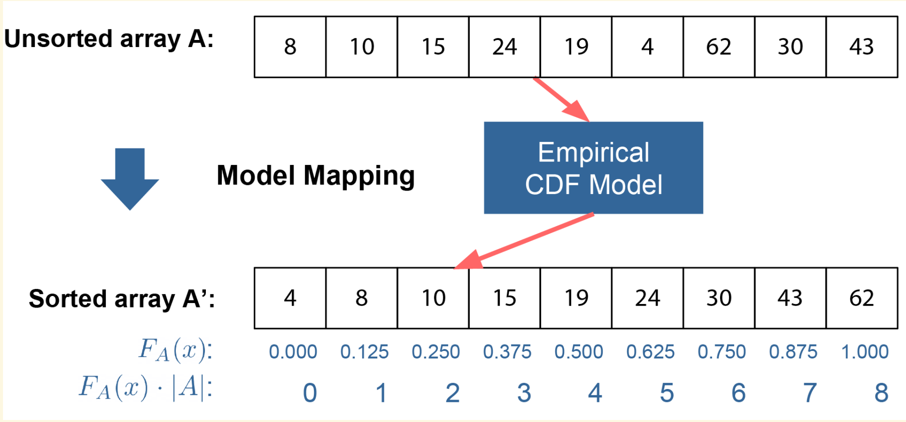
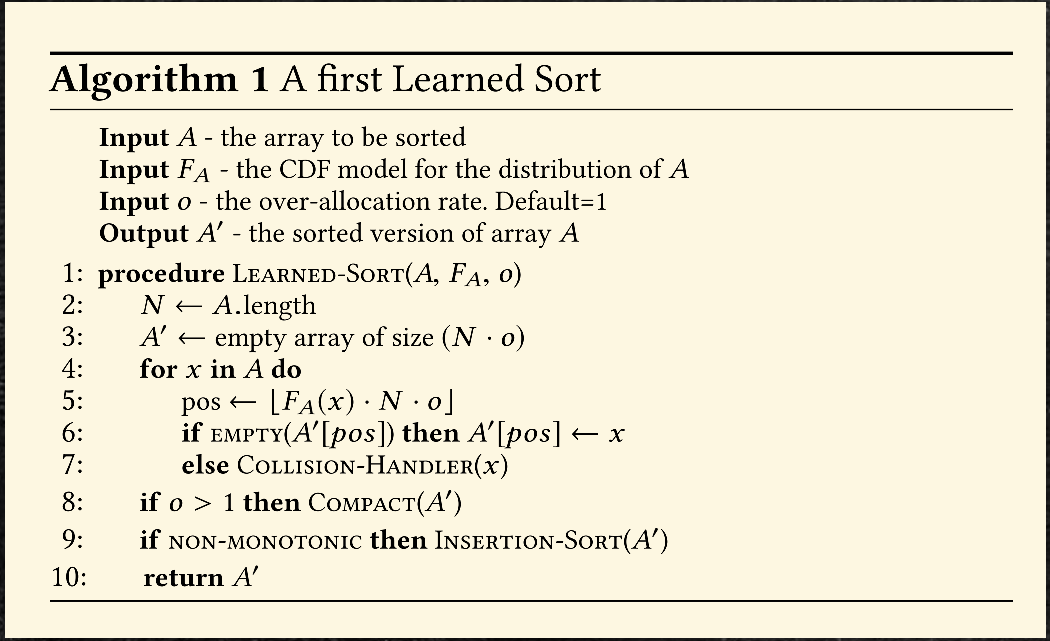


Figure 1: Sorting with the perfect CDF model

- But even in the simplest case (where CDF is identity), took 38.7 sec (vs Radix Sort 37.5 sec)
- Reason? Cache misses

First Learned Sort



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 $A'$
```

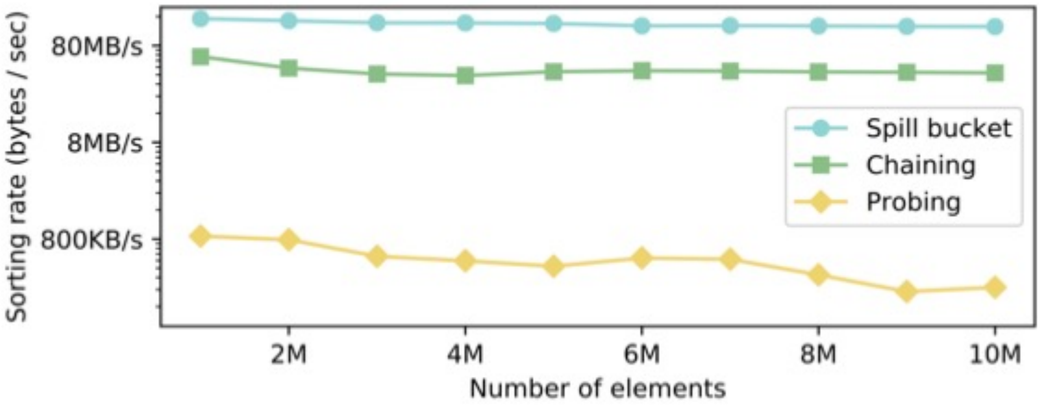


Figure 2: The sorting rate for different collision handling strategies for Algorithm 1 on normally distributed keys.

Collision handling: (1) Scan the array for closest empty slot; (2) Do chaining; (3) Spill bucket

First Learned Sort

- Need to overfit the data to get lowest collisions
  - But need to work with a sample to keep costs low

- Can "over-provision" (i.e., use a larger target array)
  - Extra space requirements and more cache misses
  - Also, need to deal with gaps at the end

- Do "bucketing", i.e., map each item to a bucket than a specific position
  - Smaller range, so can be more accurate
  - Need to recursively sort the buckets

Cache-optimized Radix Sort

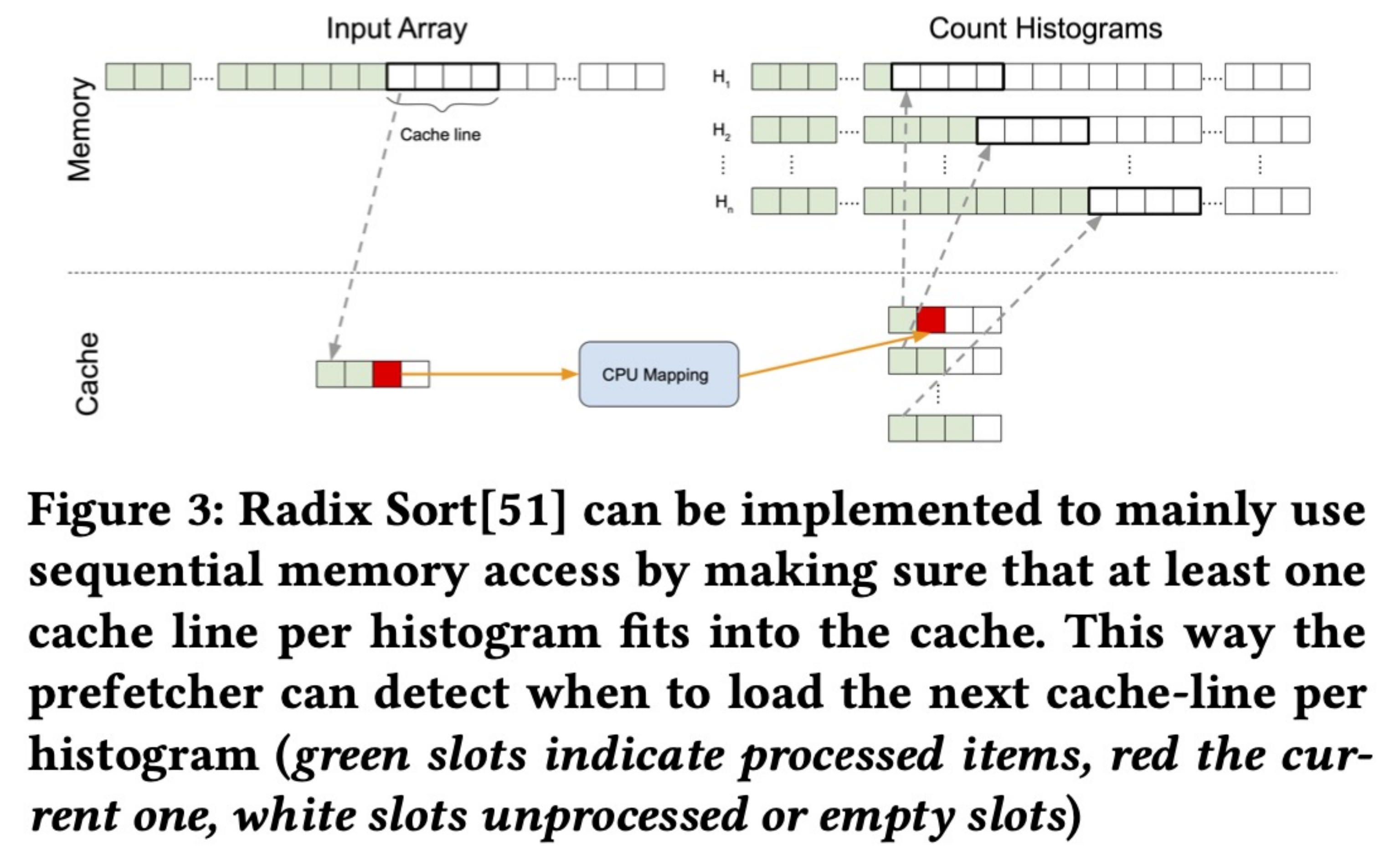


Figure 3: Radix Sort[51] can be implemented to mainly use sequential memory access by making sure that at least one cache line per histogram fits into the cache. This way the prefetcher can detect when to load the next cache-line per histogram (*green slots indicate processed items, red the current one,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

CDF-based Sort

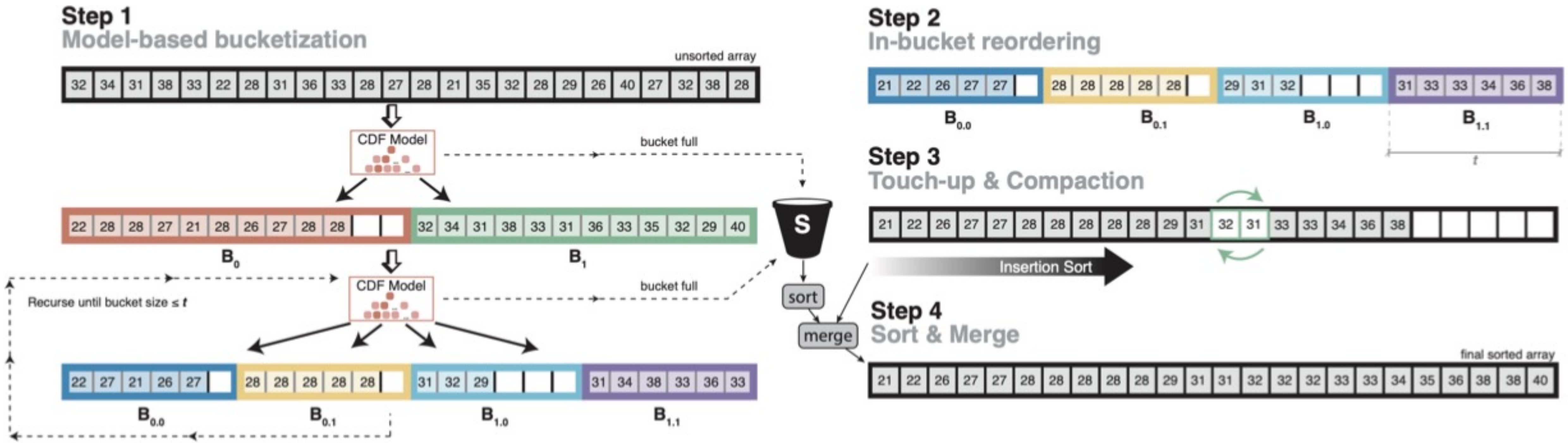


Figure 4: Cache-optimized Learned Sort: First the input is partitioned into f fixed-capacity buckets (here $f = 2$) and the input keys are shuffled into these buckets based on the CDF model's predictions. If a bucket gets full, the overflowing items are placed into a spill bucket S. Afterwards, each 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).

CDF-based Sort

- Run-time almost identical to Radix Sort for dense keys
  - i.e., key domain size is close to the data size
  - But better as data size << key domain size -- each pass does more in CDF-based sort
- Choice of CDF being learned
  - Recursive Model Index
  - Used spline fitting instead of linear regression to avoid overlaps

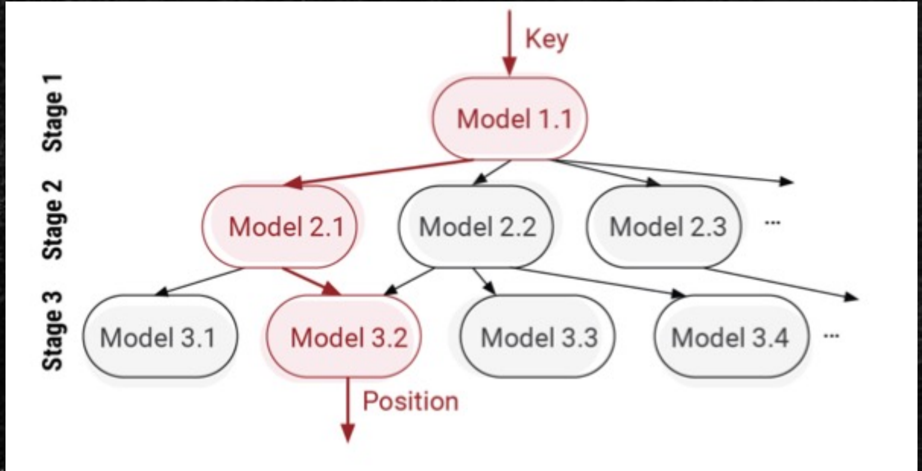
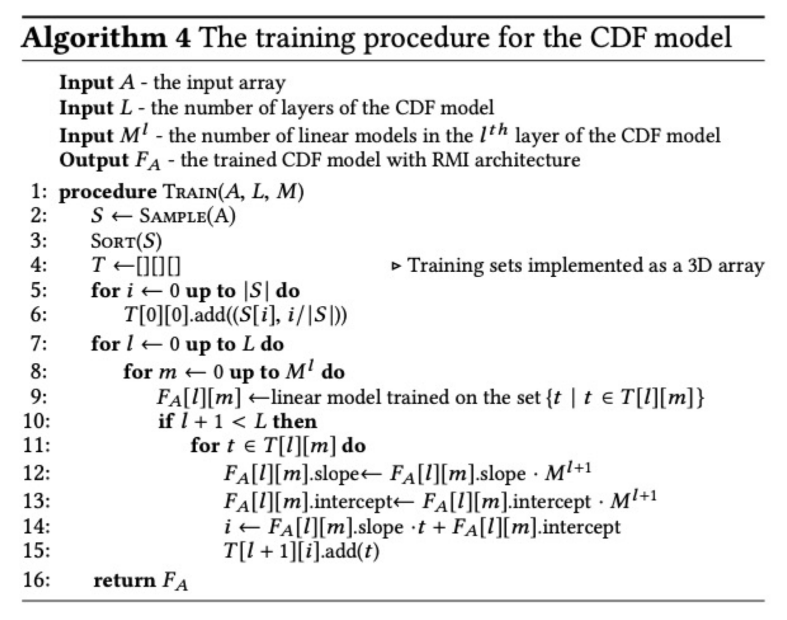


Figure 5: A typical RMI architecture containing three layers



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
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 $F_A$
```

Experiments

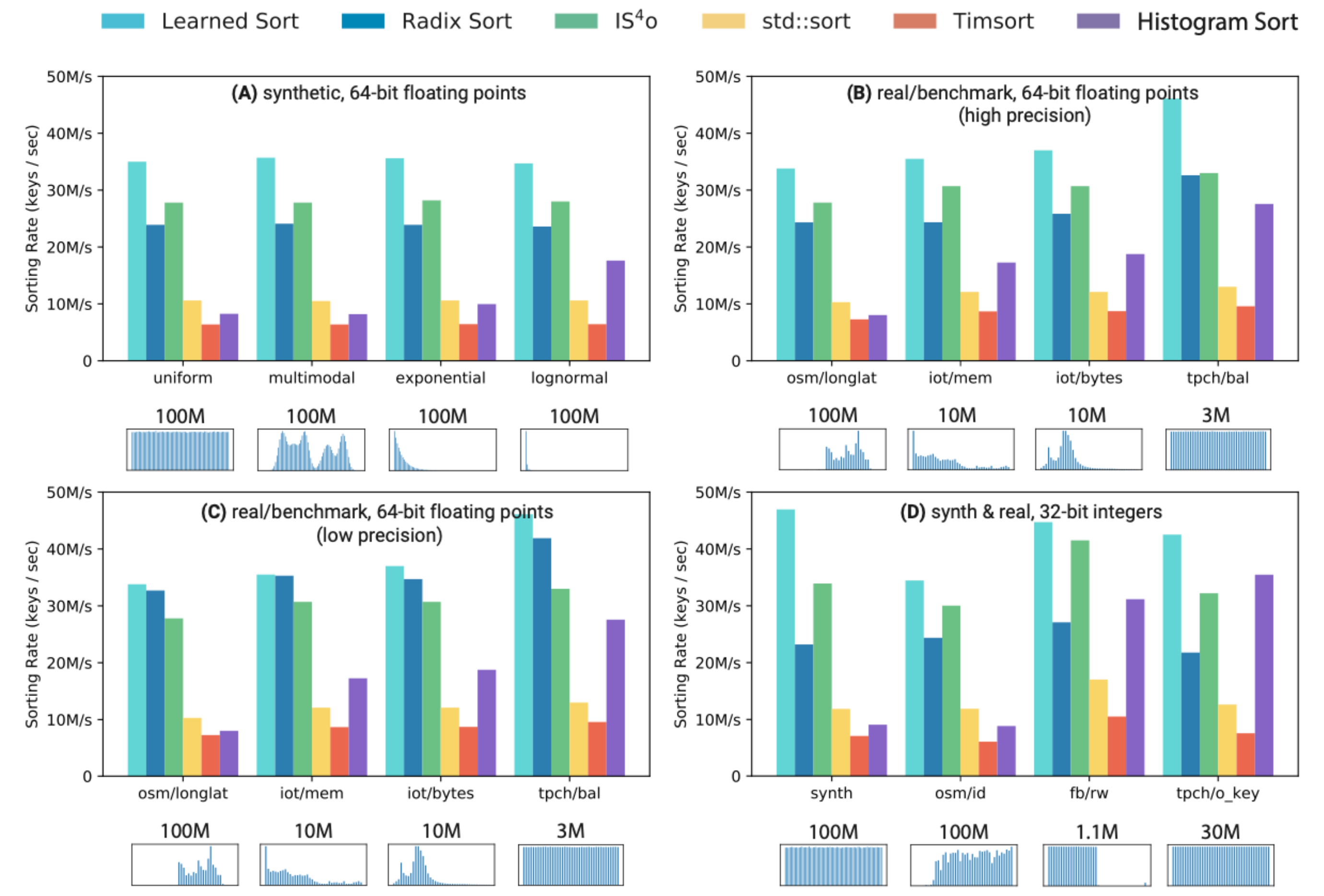


Figure 9: The sorting rate of Learned Sort and other baselines for real and synthetic datasets containing both doubles and integers. The pictures below the charts visualize the key distributions and the dataset sizes.

Experiments

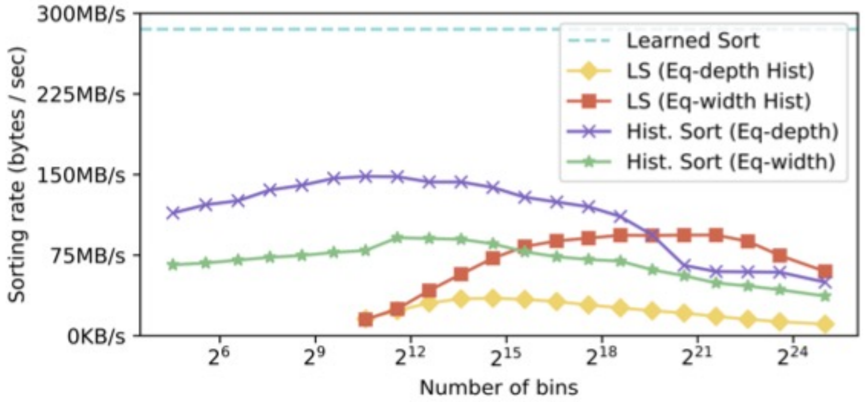


Figure 14: The sorting rate of Learned Sort algorithm on 100M normally-distributed keys as compared with (1) a version of LS that uses an equi-depth histogram as CDF model, (2) a version with an equi-width histogram, (3) Equi-depth Histogram Sort, and (4) Equi-width Histogram Sort.

Outline

- Sorting
- **Joins**

Index Nested Loops Join

- Assumes that there is an existing "index" on the inner relation
- Can we use RMI as that index?
  - Do we assume one exists?
- Not efficient to use as is – instead use a "gapped" version

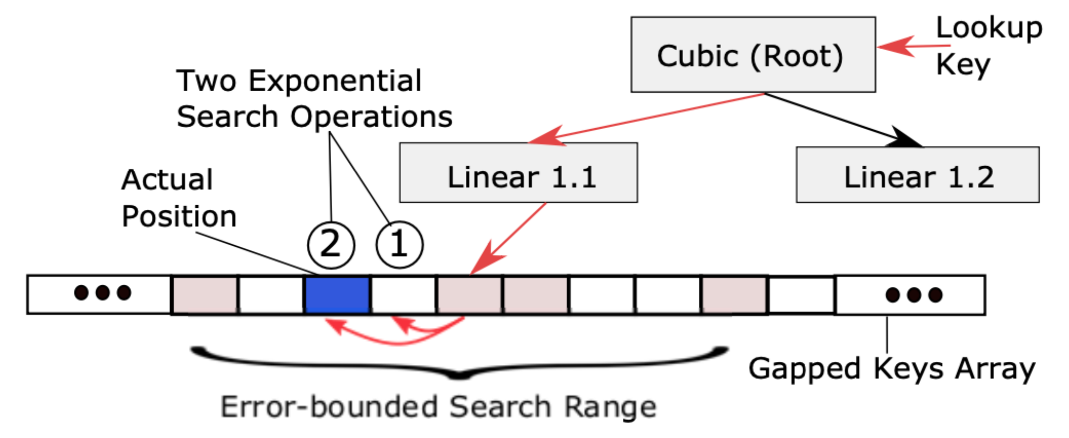


Fig. 3. Example of 2-levels Gapped RMI (GRMI)

Index Nested Loops Join

- Too many cache misses
- Instead, ”buffer” requests

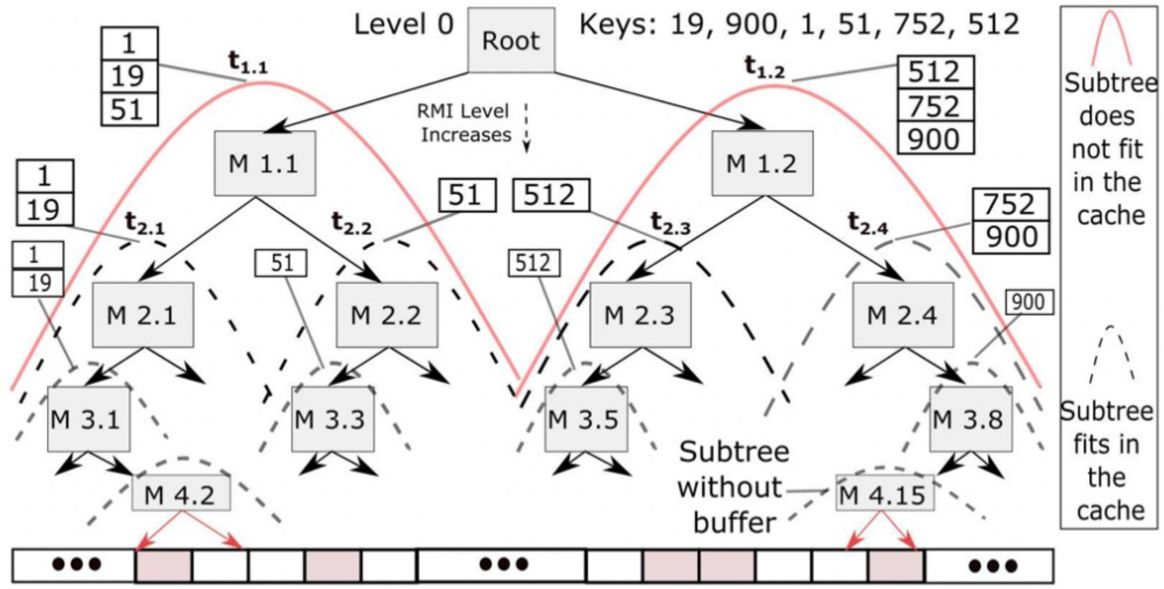
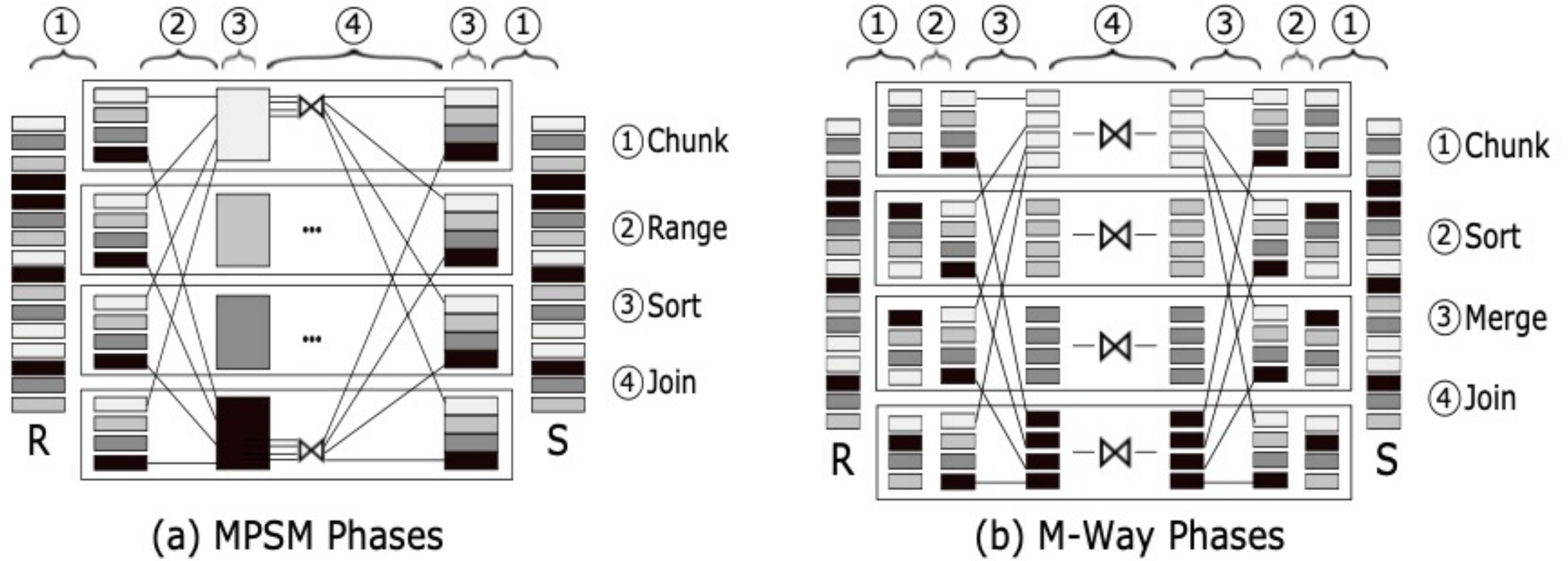


Fig. 4. Example on hierarchical Request Buffers at some RMI models.

Sort-Merge Join

- Prior techniques for multi-core sorts



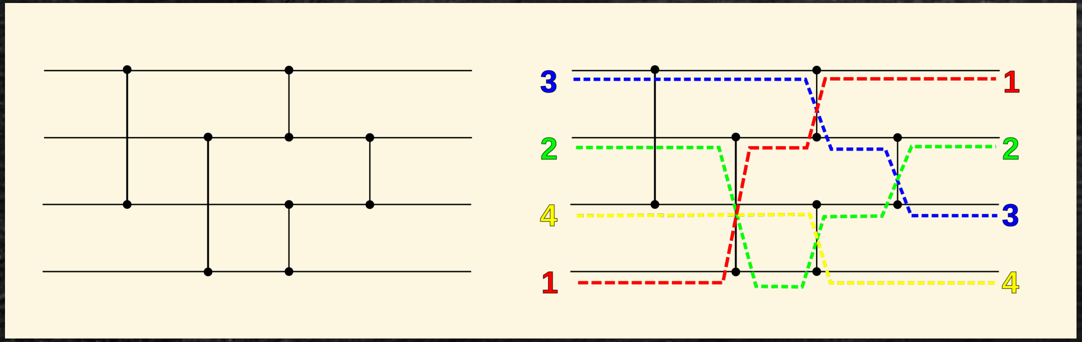
(a) MPSM Phases

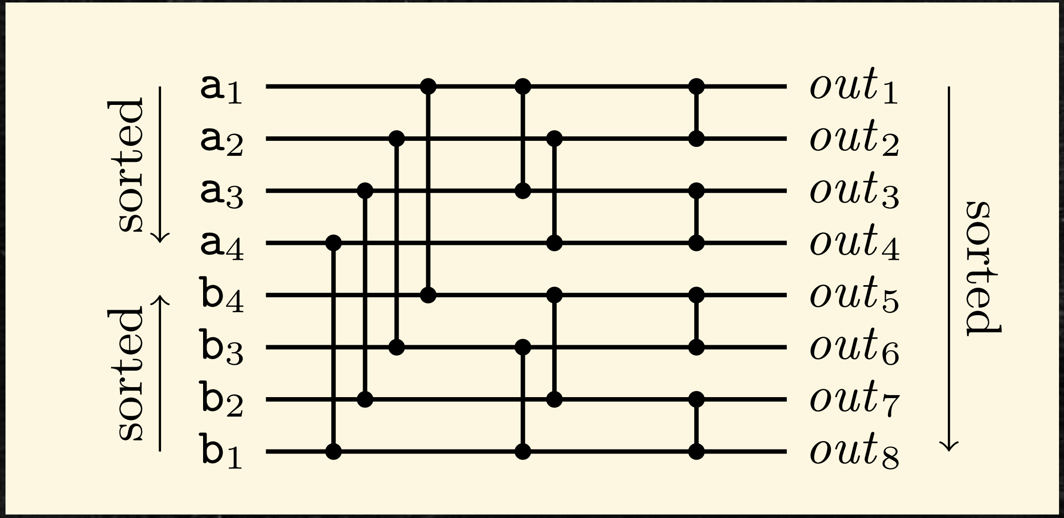
(b) M-Way Phases

Fig. 1. Sort-based Joins: MPSM and MWAY

Sort-Merge Join

- Bitonic Sort





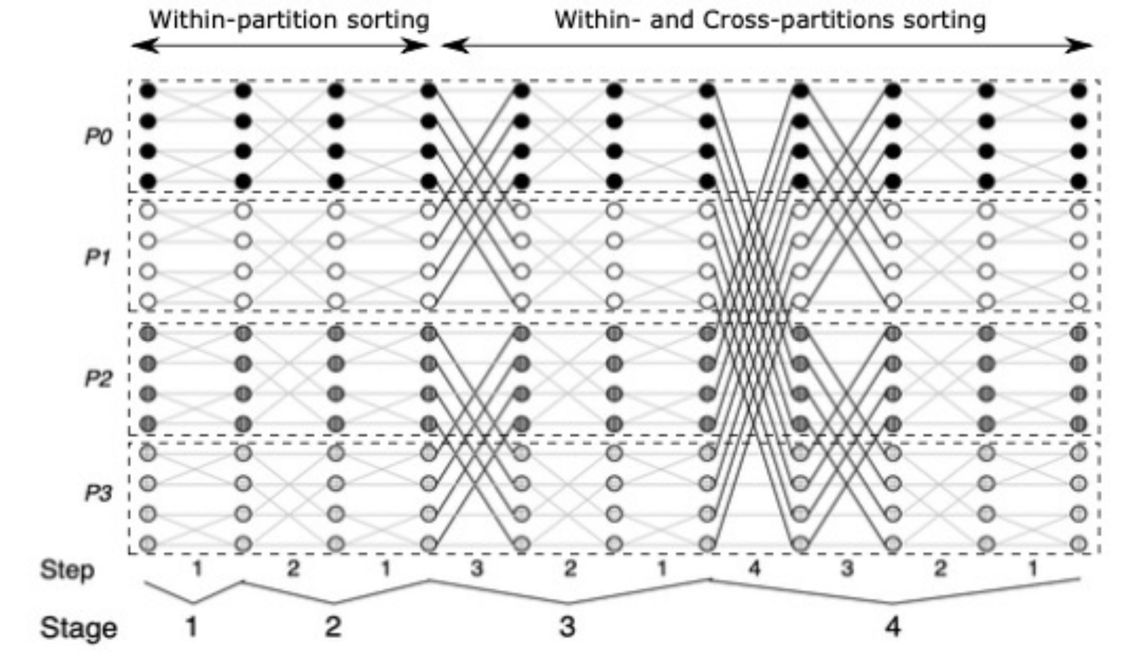


Fig. 7. Example of a bitonic sorting network for 16 items with 4 partitions.

Learned SMJ

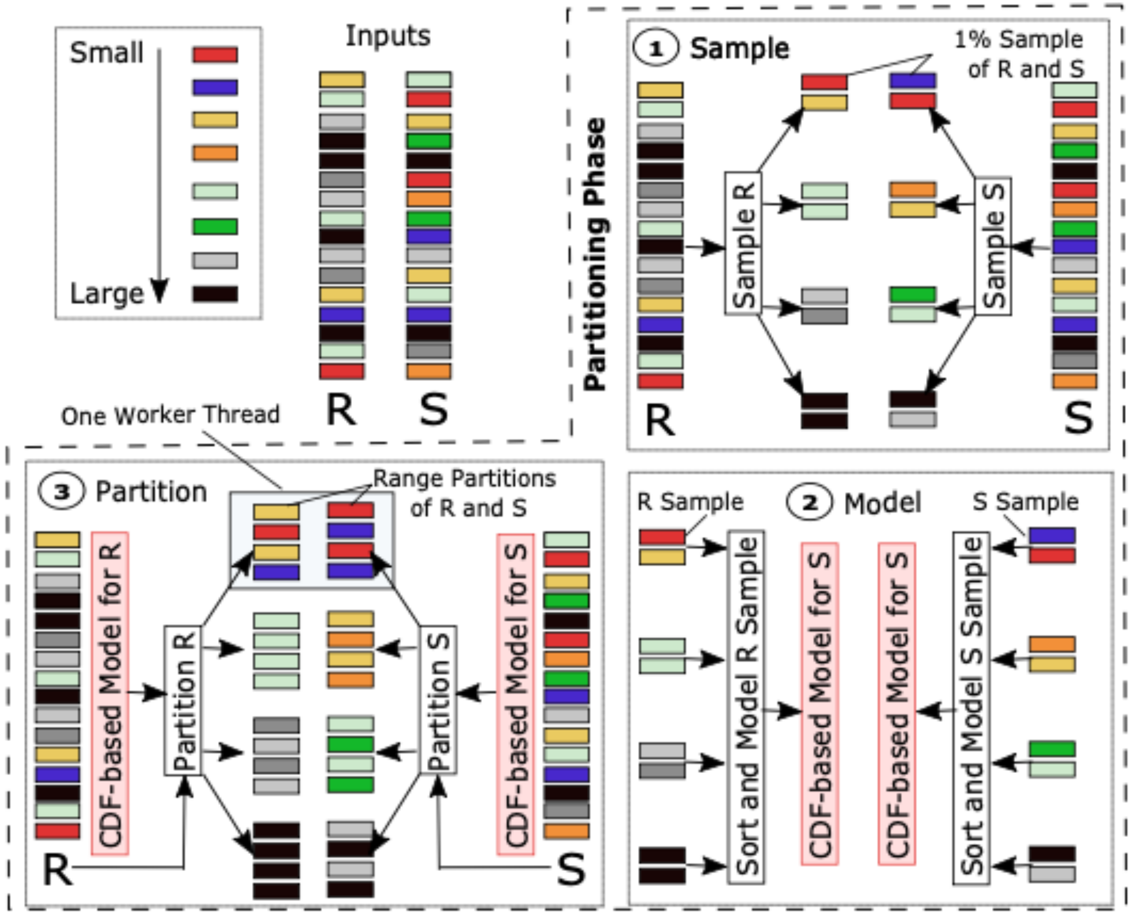


Fig. 5. L-SJ partitioning phase (the color of a key reflects how large it is).

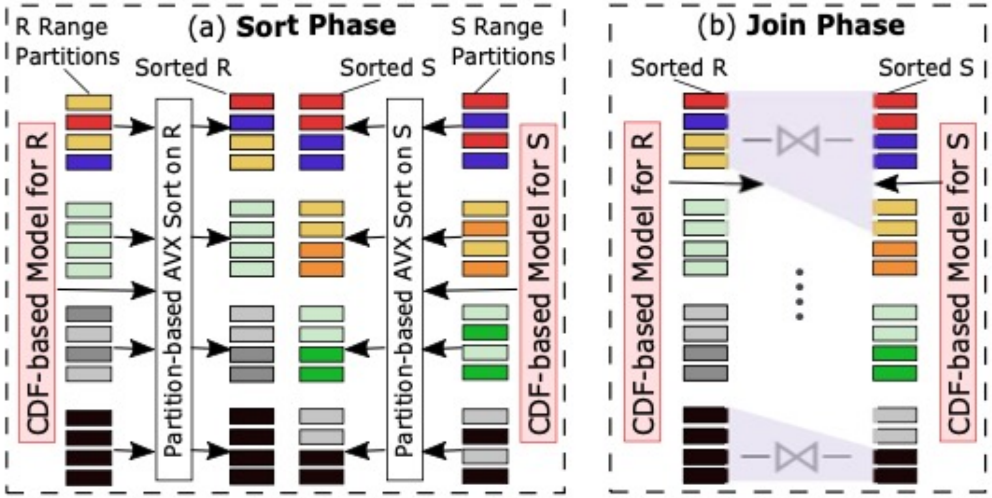


Fig. 6. L-SJ sorting and joining phases (Continued example from Figure 5).

Experiments

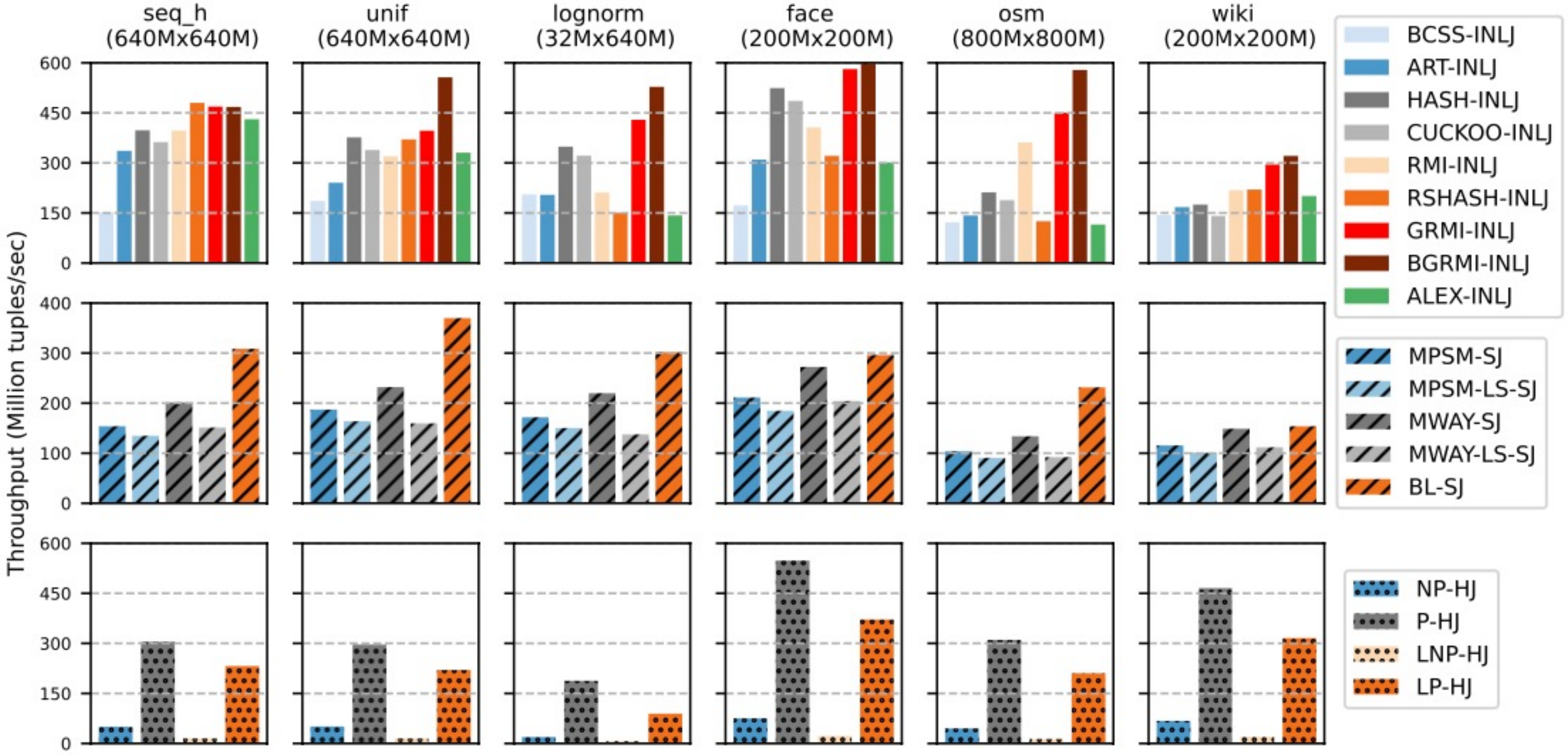


Fig. 8. Performance of the three join categories for real and synthetic datasets, where each row represents a category.

Some Discussion Points

- What’s the main take-away from this paper?
- Major concerns with the paper?
- Possible improvements?