Multi-Way Co-Ranking: Index-Space Partitioning of Sorted Sequences Without Merge

Amit Joshi
amitjoshi2724@gmail.com
amit.joshiusa@gmail.com
Independent Researcher

Abstract

We present a merge-free algorithm for *multi-way co-ranking*, the problem of computing *cut indices* $i_1, \ldots, i_m$ that partition each of the m sorted sequences such that all prefix segments together contain exactly K elements. Our method extends Siebert and Träff's [3] [2] two-list co-ranking to arbitrary m, maintaining per-sequence bounds that converge to a consistent global frontier without performing any multi-way merge or value-space search. Rather, we apply binary search to *index-space*. The algorithm runs in $O(\log(\sum_t n_t)\, \log m)$ time and $O(m)$ space, independent of K. We prove correctness via an exchange argument and discuss applications to distributed fractional knapsack, parallel merge partitioning, and multi-stream joins.

Keywords: Co-ranking, partitioning, Merge-free algorithms, Index-space optimization, Selection and merging, Data structures

1. Introduction

Problem.. Given m sequences $L_1, \ldots, L_m$, each sorted in non-decreasing order (duplicates allowed) by convention and a global target rank $K \in \{0, \ldots, N\}$ with $N = \sum_t n_t$, the *multi-way co-ranking* problem asks for cut indices $i_1, \ldots, i_m$ such that:

$$\sum_{t=1}^{m} i_t = K \quad \text{and} \quad \max_t \ell_t \;\le\; \min_t r_t,$$

where $\ell_t = (i_t > 0)\,?\, L_t[i_t - 1] : -\infty$ and $r_t = (i_t < n_t)\,?\, L_t[i_t] : +\infty$ (sentinels at ends). Intuitively, the first i_t items of L_t are part of a global K-prefix (up to ties).

Contributions.. We introduce a merge-free, *index-space* algorithm that generalizes classic two-list co-ranking to arbitrary m:

- We design an iterative **donor/receiver** greedy scheme driven by *max-left* (largest ℓ_t) and *min-right* (smallest r_t) respectively and move mass between the two with the invariant that total mass always equals K. We store explicit upper and lower bounds for each list $Lb_t \le i_t \le Ub_t$.
- We show that **halving** the lower or upper range of the donor or receiver during each iteration yields fast convergence ($Lb_t = i_t = Ub_t$) in just

$$O\Big(\log(\sum_t n_t)\Big) \text{ iterations.}$$

- We use **indexed (addressable) heaps** to efficiently query ℓ_t and r_t each iteration, and achieve a clean complexity bound independent of K:

$$O\Big(\log(\sum_t n_t) \cdot \log m\Big) \text{ time}, \qquad O(m) \text{ space.}$$

- We give an **exchange-style correctness** proof: among all feasible infinitesimal transfers, moving mass from max-left to min-right achieves maximal progress on a natural imbalance potential, and any optimal correction sequence can be reordered so extremal transfers happen first without loss.
- We discuss **applications**: distributed fractional knapsack (multi-source), parallel merge partitioning, and multi-stream join partitioning.

Why index space?. Many selection approaches reason in value space: merge the streams, maintain a heap of cursor values, or biselect a numeric threshold. Our method never merges or searches the value domain; it reasons *only* on indices and the relative order that the sorted property guarantees. This aligns with co-ranking's roots [2] and leads to a deterministic, K-independent complexity, just like in [3].

2. Background and Related Work

Two-list co-ranking (partitioners for merge).. Siebert and Traff's [2] *co-ranking* for two sorted inputs (used to partition work in parallel merge) finds (j, k) s.t. $j + k = i$ and the order constraints across the merge frontier hold. Efficient implementations do a binary-search-like *halving* of the feasible interval, yielding $O(\log n + \log m)$ time for the i-th output frontier. Siebert later used 2-way co-ranking work in [3] to create an in-place merge-sort, but again among only 2 sorted sequences. Our work *generalizes the partitioner* from 2-way to m-way, in index space and neatly describes the changes needed to do so.

Splitter-based parallel sorting (Siebert & Wolf).. Siebert and Wolf [4] introduced an exact splitter-finding method for scalable parallel sorting. Their algorithm computes $p - 1$ *splitter values* that partition the global key space so each process receives an equal share of elements, achieving $O(\frac{n}{p} \log n + p \log^2 n)$ time. This approach operates entirely in *value space*: it searches for key thresholds that balance partitions during redistribution. By contrast, our multi-way co-ranking algorithm operates in *index space* on already-sorted sequences and directly outputs the

co-rank vector $(i_1, \ldots, i_m)$ satisfying $\sum i_t = K$ and $\max \ell_t \leq \min r_t$, without any global value search. Splitters compute thresholds; co-ranking computes frontier indices and partitions sequences into prefixes and suffixes independent of values.

Selection in sorted partitions (Frederickson–Johnson).. A classical line of work due to Frederickson and Johnson [1] studies *selection* and *ranking* over structured inputs such as the union of m sorted lists ("sorted partitions"). Their algorithms are fundamentally *value-space* procedures: they maintain numeric thresholds (candidate keys), repeatedly count how many elements are $\leq$ a threshold across the lists (via one-sided searches), and narrow the feasible value interval until the K-th *value* is identified. In the multi-list setting this yields bounds of the form $O(m + \sum_{t=1}^{m} \log n_t)$ for selecting the K-th smallest from m sorted lists, with closely related results for ranking a given key [1]. By contrast, our contribution is an *index-space* primitive: we do not search the key domain or perform global "count-$\leq$" operations. Instead, we directly evolve the per-sequence *cut indices* $(i_1, \ldots, i_m)$ satisfying $\sum_t i_t = K$ and the co-rank frontier condition, using only local comparisons at the current indices. Thus, while both approaches address "K-th from multiple sorted sources"–type problems, Frederickson–Johnson output a *value* via thresholding, whereas we output the full *co-rank vector* without value bisection.

This paper's positioning.. We treat *multi-way co-ranking* as a merge-free *partitioning primitive* that is an efficient and simple generalization of 2-way co-ranking described in [2]. The novelty is methodological (efficient indexed heaps for fast updates, explicit upper and lower bounds) and conceptual (the dynamics of halving the feasible range, operating in index-space), with applications in fast queries in distributed environments, databases, and parallel partitioning and/or merging of data.

3. Preliminaries and Notation

We have m sorted sequences L_t of lengths n_t, total $N = \sum_t n_t$, and a target rank $K \in [0, N]$. We maintain an index vector $i = (i_1, \ldots, i_m)$ with feasibility *bounds*

$$Lb[t] \leq i[t] \leq Ub[t], \qquad \sum_t i[t] = K.$$

Lower bounds Lb are initialized at 0. Upper bounds Ub are initialized at n_t. We define left/right boundary values with sentinels on the ends:

$$\ell_t = \begin{cases} (-\infty, t) & i_t = 0 \\ (L_t[i_t - 1], t) & \text{else} \end{cases} \qquad r_t = \begin{cases} (+\infty, t) & i_t = n_t \\ (L_t[i_t], t) & \text{else} \end{cases}$$

The co-rank condition is $\max_t \ell_t \leq \min_t r_t$. In cases of ties, we adopt a simple canonical convention: lower list indices are treated as "smaller," so this ordering applies consistently in both the min-heap and max-heap. For the max-heap, this means that if two boundary values are equal, the element from the later list index is considered "larger" and will be returned first. This tie-breaking rule is purely conventional and ensures deterministic behavior; if the specific ordering of equal elements is irrelevant to an application, the tie-handling condition may be omitted altogether. If the co-rank condition is not satisfied, we select a donor p and receiver q. We define the donor slack to be $i[p] - Lb[p]$ — how much p can donate before hitting its lower bound. Similarly, we define the receiver "slack" to be $Ub[q] - i[q]$ how much mass q can receive before hitting its upper bound. We tighten at least one of these bounds on every iteration after the first one.

4. Data Structure: Indexed (Addressable) Heaps

We maintain two indexed heaps over the fixed ID set $\{0, \ldots, k-1\}$:

- HL: a *max-heap* on keys (ℓ_t, t), returning the donor p with largest ℓ_t; on elemental ties, larger t first (or your chosen tie).
- HR: a *min-heap* on keys (r_t, t), returning the receiver q with smallest r_t; on elemental ties, smaller t first.

Each supports $O(\log m)$ `update_key(id,new_key)` and $O(1)$ `peek`. Because indices change only for at most two lists per round (donor and one receiver), each round incurs $O(\log m)$ heap work.

Algorithm 1 Indexed heap (interface summary)

```
insert(id, key)                          ▷ O(log m)
update_key(id, newkey)                   ▷ O(log m) sift up/down
peek() → (key, id)                       ▷ O(1)
```

5. Algorithm

High-level idea.. Start with any feasible i (can water-fill the first however many lists) such that $\sum_{t=1}^{m} i_t = K$. If $\max \ell \leq \min r$ we are done. Otherwise, choose donor $p = \arg\max \ell$ and receiver $q = \arg\min r$. Consider the smaller of the two slacks (for example, if the donor has less room than the receiver, then select that), halve that slack, and transfer that mass from the donor to the receiver. This ensures that both the donor and receiver stay in their respective feasible zones. Update bounds and heap keys and repeat. Because we always transfer from the worst left boundary to the best right boundary, a simple potential function decreases monotonically. We describe our algorithm in detail in Algorithm 2.

6. Correctness

We sketch a concise correctness argument. Let $\Phi(i) = \max_t \ell_t - \min_t r_t$ denote the imbalance potential; the algorithm terminates when $\Phi \leq 0$.

Lemma 1 (Local extremal optimality)**.** *If $\Phi(i) > 0$, let $p = \arg\max_t \ell_t$ and $q = \arg\min_t r_t$. Among all feasible infinitesimal transfers of index mass that preserve $\sum_t i_t = K$, the pair (p, q) achieves the greatest non-increasing change in Φ. Any other pair (u, v) can produce at most the same or a weaker decrease in Φ.*

Proof sketch. Decreasing i_p reduces ℓ_p (the local maximum on the left boundary), and increasing i_q raises r_q (the local minimum on the right boundary). Since $\ell_p \geq \ell_u$ and $r_q \leq r_v$ for all lists u, v, the extremal pair (p, q) produces the steepest possible reduction of the gap $\max \ell - \min r$. Tie handling through the heap comparator (lower index first on the right, higher index first on the left) deterministically resolves symmetry but does not affect monotonicity of Φ. □

Lemma 2 (Exchange of transfer order)**.** *Any finite sequence of feasible transfers that decreases Φ can be reordered so that all extremal (p, q) transfers occur before any non-extremal transfer, without worsening Φ at any intermediate step.*

Proof sketch. Each transfer modifies only two indices (i_u, i_v) and can affect Φ only through the associated ℓ_u or r_v. By the previous lemma, the extremal pair has dominant effect: executing it earlier cannot increase Φ. Commuting a non-extremal move past an extremal one therefore preserves or improves the potential. Repeatedly commuting yields a sequence in which all extremal moves occur first and Φ is weakly smaller at every stage. □

Theorem 1 (Convergence and validity)**.** *Algorithm 2 terminates with a valid co-rank vector $(i_1, \ldots, i_m)$ satisfying $\sum_t i_t = K$ and $\max_t \ell_t \leq \min_t r_t$.*

Proof sketch. Feasibility ($L_b \leq i \leq U_b$ and $\sum i_t = K$) is maintained by construction: each update transfers exactly Δ units of index mass from a donor p to a receiver q and tightens their respective bounds ($U_b[p] \leftarrow i_p$, $L_b[q] \leftarrow i_q$). At each round the potential Φ strictly decreases, because the move reduces either ℓ_p or r_q (whichever side was limiting), or else a bound becomes saturated and is excluded from further motion. Since both L_b and U_b are monotone and integer-valued, the process admits only finitely many distinct states and must terminate. By the exchange lemma, the greedy extremal choice made in each round is at least as effective as any alternative sequence. Upon termination $\Phi \leq 0$, which by definition implies $\max_t \ell_t \leq \min_t r_t$. □

7. Complexity Analysis

Let $n_t = |L_t|$ and $N = \sum_t n_t$. We outline the cost components.

Rounds.. In each round, at least one side—either the donor or the receiver—is halved, whichever has the smaller available distance to its bound. If the donor p has the smaller distance, its slack $s_p = i_p - Lb[p]$ is halved ($i_p \leftarrow i_p - \lceil s_p/2 \rceil$ and $Ub[p] \leftarrow i_p$); otherwise, the receiver q has its slack/headroom $h_q = Ub[q] - i_q$ halved ($i_q \leftarrow i_q + \lceil h_q/2 \rceil$ and $Lb[q] \leftarrow i_q$). Each list's distance to feasibility, $U_b[t] - L_b[t]$, can therefore shrink only $O(\log n_t)$ times before stabilizing. Aggregating across all m lists and observing that the global potential $\Phi = \sum_t (U_b[t] - L_b[t])$ decreases geometrically, the total number of rounds is

$$T = O\Big(\log\Big(\sum_{t=1}^{m} n_t\Big)\Big).$$

Since each round performs a constant number of indexed heap operations ($O(\log m)$ time), the overall runtime is $O(\log(\sum_t n_t) \log m)$, with linear (in the number of sequences) $O(m)$ space.

Total.. Combining, the time is

$$\boxed{O\Big(\log\big(\textstyle\sum_t n_t\big) \cdot \log k\Big)}, \qquad \text{space } \boxed{O(k)}.$$

8. Applications

Distributed fractional knapsack (multi-source).. The fractional knapsack problem is a variation of the classic knapsack (take some jewels such that their total value is maximized yet their cumulative weight fits in a given capacity) problem where taking a fraction of a jewel in your knapsack is allowed. This admits a greedy solution, where we sort jewels in decreasing order by density (value per weight), and select the highest density jewels. When taking the next jewel would put our knapsack over capacity, we cut that jewel so that its weight fits with the others under a total capacity, and add the proportional value to our score.

When a fractional knapsack must draw from m sources (shards) already individually sorted (in non-increasing order) by density, one can compute the global K-prefix split (at a suitable discretization of mass) without merging sources. We can binary search on the largest K, such that the prefix segments have cumulative weight within capcity. For these efficient prefix sum queries, we can use an augmented (with sum) self-balancing BST or perhaps a similar advanced Linked List that allows for $O(\log(n))$ queries, insertions, and deletions if each source is dynamic.

Parallel m-way merge partitioning.. Assign processors disjoint prefixes without performing a preliminary merge. The co-rank vector determines exact hand-offs; processors then perform local merges on their ranges only. This would essentially be Siebert and Wolfs' work [4] with a different splitting mechanism – one that operates in *index-space* rather than binary searching on value splitters.

Multi-stream join partitioning (Databases).. In database or stream-processing pipelines, partitioning the join frontier at a global rank is natural; our method yields the per-stream cursors consistent with the global prefix.

9. Conclusion

We presented an index-space, merge-free algorithm for *multi-way co-ranking* that generalizes two-way co-ranking to arbitrary m-ways. The method uses explicit lower and upper bounds for each sequence, halving of feasible ranges, and an invariant that the cut indices for each partition add up to K, yielding a clean $O(\log(\sum n_t) \cdot \log m)$ time and $O(m)$ space bound, independent of K (the global rank being queried). The approach is simple to implement, proof-friendly, and directly applicable to distributed optimization and parallel partitioning and merge tasks. Finally, as a thanks for reading, we release our implementation code.[1]

References

[1] Greg N. Frederickson and Donald B. Johnson. Generalized selection and ranking. In *Proc.* 12th Annual ACM Symposium on Theory of Computing (STOC), *New York, NY, USA*, pages 420–428, 1980. doi: 10.1145/800141.804690. Complexity bound $O(k + \sum_{i=1}^{k} \log n_i)$ for union of k sorted lists.

[2] Christian Siebert. Perfectly load-balanced, stable, synchronization-free parallel merge. *Parallel Processing Letters*, 24(01):1450005, 2014. doi: 10.1142/S0129626414500054. URL https://www.worldscientific.com/doi/abs/10.1142/S0129626414500054.

[3] Christian Siebert. Simple in-place yet comparison-optimal mergesort, 2025. URL https://arxiv.org/abs/2509.24540. arXiv preprint arXiv:2509.24540.

[4] Christian Siebert and Felix Wolf. A scalable parallel sorting algorithm using exact splitting. RWTH Aachen University technical report, RWTH-CONV-008835, 2011. URL https://publications.rwth-aachen.de/record/50880/files/50880.pdf.

[1] https://github.com/amitjoshi2724/multi-way-co-ranking

Algorithm 2 Multi-Way Co-Ranking

Require: Sorted sequences $L_1, \ldots, L_m$ (ascending), lengths $n_t = |L_t|$, target rank $K \in [0, \sum_t n_t]$.

Ensure: Indices $i_1, \ldots, i_m$ with $\sum_{t=1}^{m} i_t = K$ and $\max_t \ell_t \leq \min_t r_t$.

Bounds: $L_b[t] \leftarrow 0$ for all $t \in \{1, \ldots, m\}$; $U_b[t] \leftarrow n_t$.
Init feasible i: set $i_t \leftarrow L_b[t]$; `need` $\leftarrow K$.
for $t = 1$ **to** m **do** ▷ water-fill up to K using capacities $(U_b[t] - i_t)$
  `cap` $\leftarrow U_b[t] - i_t$; `take` $\leftarrow \min\{$`cap`, `need`$\}$;
  $i_t \leftarrow i_t +$ `take`; `need` $\leftarrow$ `need` $-$ `take`; **if** `need` $= 0$ **then break**
end for

Boundary access:
$\ell_t \leftarrow \begin{cases} -\infty & \text{if } i_t = 0 \\ L_t[i_t - 1] & \text{otherwise} \end{cases}$ and $r_t \leftarrow \begin{cases} +\infty & \text{if } i_t = n_t \\ L_t[i_t] & \text{otherwise.} \end{cases}$
Heaps: Build an *indexed* max-heap HL keyed by ℓ_t (ties $\rightarrow$ larger t first), and an indexed min-heap HR keyed by r_t (ties $\rightarrow$ smaller t first).
for $t = 1$ **to** m **do**
  HL.`insert`(t, ℓ_t); HR.`insert`(t, r_t)
end for

for round $= 1, 2, \ldots$ **do**
  $(x, p) \leftarrow$ HL.`peek()` ▷ donor candidate: max-left
  $(y, q) \leftarrow$ HR.`peek()` ▷ receiver candidate: min-right
  if $x < y$ **or** $(x = y \ \wedge \ p \leq q)$ **then** ▷ authoritative stop condition
    return $i_1, \ldots, i_m$
  end if

  Halving magnitudes on both sides:
  $\Delta_p \leftarrow \left\lceil \frac{i_p - L_b[p]}{2} \right\rceil$ ▷ donor slack halved
  $\Delta_q \leftarrow \left\lceil \frac{U_b[q] - i_q}{2} \right\rceil$ ▷ receiver headroom halved
  $\Delta \leftarrow \min\{\Delta_p, \Delta_q\}$ ▷ move at most what both sides can support
  if $\Delta = 0$ **then**
    (degenerate case) break ▷ cannot progress; should not occur if not done
  end if

  Tighten explicit bounds at the endpoints of the move:
  $U_b[p] \leftarrow i_p$ ▷ donor cannot exceed its current index this round
  $L_b[q] \leftarrow i_q$ ▷ receiver cannot go below its current index this round

  Commit the balanced transfer and refresh heap keys:
  $i_p \leftarrow i_p - \Delta$; $i_q \leftarrow i_q + \Delta$
  Update ℓ_p, r_p and ℓ_q, r_q; HL.`update_key`(p, ℓ_p), HR.`update_key`(p, r_p)
  HL.`update_key`(q, ℓ_q), HR.`update_key`(q, r_q)
end for