DeltaSort: Incremental repair of sorted arrays with known updates

Shubham Dwivedi
Independent Researcher
shubd3@gmail.com

2026

Abstract

Reading sorted data is a fundamental requirement in almost all data processing systems. When dealing with large sorted datasets that require great read performance, sorting-on-read is not feasible. A standard approach is to have a derived sorted read-replica that is updated with the latest snapshot asynchronously whenever the system-of-record gets updated. For updating read-replicas, most production systems resort to either full re-sorting or Binary-Insertion-Sort or Extract-Sort-Merge. In this paper, we present how we can do better than the traditional approaches for certain workloads. This paper also formulates an alternative model in which the sorting routine is explicitly informed of the updated indices since the previous sort. Under this model, we present *DeltaSort*, an efficient incremental repair algorithm for arrays. We present experimental evidence that shows that *DeltaSort* outperforms existing approaches for a wide range of update batch sizes for our Rust implementation. These results suggest that tighter integration between update pipelines and sorting routines can yield significant performance gains in real incremental-sorting workloads. We also explore limitations of this approach and identify a clear crossover point, depending on array size, where full re-sorting becomes preferable.

1 Introduction

Sorting is among the most optimized primitives in modern systems, backed by decades of deep research. Standard library implementations—TimSort [13], Introsort [10], and PDQSort [12] deliver excellent performance for general inputs by exploiting partial order, cache locality, and adaptive strategies. However, these algorithms operate under a *blind* model: they discover structure dynamically rather than being explicitly informed about which values have changed since the previous sort.

In many practical systems, this assumption is unnecessarily pessimistic. Sorted arrays are often maintained incrementally in read-heavy workloads where updates affect only a subset of values and the indices of those updates can be easily tracked if needed. Nevertheless, this information is typically not tracked or utilized, and systems fall back to blind re-sorting or standard repair approaches using Binary-Insertion-Sort or Extract-Sort-Merge. As an example, consider a web application that shows a real-time game leaderboard. Being able to efficiently update the sorted view on updates directly impacts the user experience. The traditional approaches do not serve us very well—full re-sort can block the main UI thread for long durations, while Binary-Insertion-Sort can be slow for a large number of updates. To address these limitations, this paper makes the following contributions:

1. **Update-aware sorting model:** We formulate an incremental sorting model in which the sorting routine is explicitly informed of the updated indices since the previous sort. Under this model, Binary-Insertion-Sort and Extract-Sort-Merge are standard baseline algorithms because they help maintain sorted order by processing incremental updates.

2. **DeltaSort algorithm:** We present *DeltaSort*, an incremental repair algorithm for sorted arrays designed for the update-aware model. DeltaSort batches multiple updates and achieves multi-fold speedups over traditional approaches for a wide range of update batch sizes.

2 Related Work

Adaptive sorting algorithms exploit existing order in the input to improve performance on nearly sorted data. TimSort [13] and natural merge sort [8] dynamically identify monotonic runs and merge them efficiently, while a substantial body of work formalizes measures of presortedness and analyzes sorting complexity as a function of these measures rather than input size alone [9]. These approaches, however, operate under a blind model: partial order must be rediscovered through full-array scans, and no external information about which values have changed is assumed.

A separate line of work studies incremental computation and view maintenance in database and streaming systems, where changes to input data are propagated to derived results using explicit delta representations [7, 11, 2]. These techniques focus on maintaining query results, aggregates, and materialized views, and operate at the level of relational or dataflow operators rather than array-based sorting primitives. While they demonstrate the practical value of update-aware computation, they do not specifically address the problem of efficiently restoring sorted order in arrays under batched point updates.

Dynamic data structures offer a different trade-off. Self-balancing trees such as AVL trees [1], red–black trees [6], B-trees [3], and skip lists [14] support efficient ordered updates with logarithmic cost, but abandon contiguous array layout and its attendant cache locality. Library sort [4] reduces insertion overhead by maintaining gaps within arrays, but addresses online insertion and incurs additional space overhead.

In contrast, this work considers maintaining sorted order in arrays where the indices of updated values since the previous sort are explicitly available. *DeltaSort* performs segmented repair without auxiliary data structures, which distinguishes it from prior adaptive sorting algorithms, incremental view maintenance techniques, and dynamic ordered data structures.

3 Problem Model

Definition 1 (Update-Aware Sorting)**.** Let $A[0..n-1]$ be an array sorted according to a strict weak ordering defined by a comparator `cmp`. Let $U = \langle u_1, u_2, \ldots, u_k \rangle$ be a sequence of indices such that

$$0 \leq u_1 < u_2 < \cdots < u_k \leq n-1,$$

and the values at these indices may have been arbitrarily updated, while values at all other indices remain unchanged. The *update-aware sorting problem* is to restore A to a state that is sorted with respect to `cmp`, given explicit knowledge of the set U.

4 DeltaSort Algorithm

4.1 Overview

Definition 2 (Violation)**.** For an updated index i, we classify its *violation* based on local order:

- **LEFT (L)**: Value **must** move left — $\texttt{cmp}(A[i-1], A[i]) > 0$ (for $i > 0$).
- **RIGHT (R)**: Value **may** move right and **cannot** move left.

Every updated index is either a LEFT or RIGHT violation. Violation is only defined for updated indices and has no meaning for clean indices.

Definition 3 (Segment)**.** A *segment* is a **maximal subarray** of U whose values follow the pattern $(\mathrm{R})^*(\mathrm{L}^+ \mid \epsilon)$: zero or more RIGHT violations followed by one or more LEFT violations, or end of array. Figure 1 illustrates this structure.

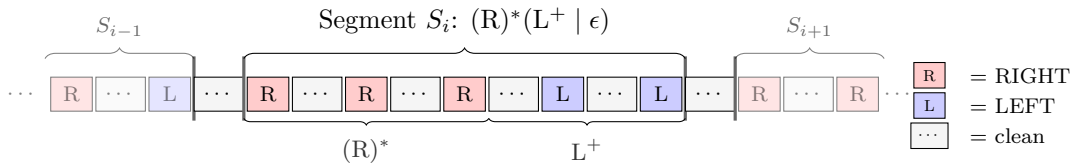

Figure 1: Each segment follows the pattern $(\mathrm{R})^*(\mathrm{L}^+ \mid \epsilon)$: zero or more RIGHT violations followed by one or more LEFT violations (or end of array).

DeltaSort operates in two phases:

1. **Phase 1 (Segment):** Extract updated values, sort them, and write back to updated indices in index order. This establishes segments in the array which are disjoint and can be repaired independently.
2. **Phase 2 (Repair):** Repair each segment left-to-right, deferring RIGHT indices to a stack until the first LEFT index is encountered. When a LEFT is encountered, first flush and repair all pending RIGHTs in LIFO order, then repair the LEFT. Continue left-to-right.

4.2 Key Insight: *Segmentation enables localized repair*

The key insight behind DeltaSort is that pre-sorting updated values induces a *segmentation* of updates. After Phase 1, updated indices partition into disjoint segments of the form $(\mathrm{R})^*(\mathrm{L}^+ \mid \epsilon)$.

Lemma 4 (Movement Confinement)**.** *Value movement during the repair phase is bounded within each segment: no value needs to cross a segment boundary.*

Proof. Let S be a segment with RIGHT indices $R_1, \ldots, R_m$ followed by LEFT indices $L_1, \ldots, L_p$ (where $m \geq 0$ and $p \geq 0$, with $m + p \geq 1$). After Phase 1, updated values are monotonically ordered by index, so $A[R_1] < \cdots < A[R_m] < A[L_1] < \cdots < A[L_p]$.

1. RIGHT values move rightward but cannot pass the first LEFT index L_1 (if it exists) or the segment boundary, since $A[R_i] < A[L_1]$ for all i.
2. LEFT values move leftward but cannot pass the last RIGHT index R_m (if it exists) or the segment boundary, since $A[R_m] < A[L_j]$ for all j.

Since no value exits its segment, segments can be repaired independently. □

4.3 Pseudocode

Algorithm 1 DeltaSort

Require: Array $A[0..n-1]$, updated indices U, comparator `cmp`
Ensure: A is sorted

Phase 1: Segment
`updatedIndices` $\leftarrow$ sort(U);
`updatedValues` $\leftarrow$ sort($[A[u] : u \in$ `updatedIndices`$]$, `cmp`)
for $i \leftarrow 0$ **to** |`updatedIndices`| $- 1$ **do**
    A[`updatedIndices`[i]] $\leftarrow$ `updatedValues`[i]
end for
`updatedIndices.push`(n); ▷ Sentinel for handling trailing segment with no LEFT's

Phase 2: Repair
`pendingRight` $\leftarrow$ []; `leftBound` $\leftarrow 0$
for $p \leftarrow 0$ **to** |`updatedIndices`| $- 1$ **do**
    $i \leftarrow$ `updatedIndices`[p]; `dir` $\leftarrow (i = n)$? LEFT : GETDIRECTION(A, i)
    if `dir` $=$ LEFT **then**
        `rightBound` $\leftarrow i - 1$
        while `pendingRight` $\neq \emptyset$ **do**
            $j \leftarrow$ `pendingRight.pop`()
            if `cmp`($A[j], A[j+1]$) > 0 **then**
                `rightBound` $\leftarrow$ FIXRIGHTVIOLATION($A, j,$ `rightBound`) $- 1$
            end if
        end while
        if $i < n$ **then** ▷ Skip dummy LEFT sentinel
            `leftBound` $\leftarrow$ FIXLEFTVIOLATION($A, i,$ `leftBound`) $+ 1$
        end if
    else
        `pendingRight.push`(i) ▷ Defer RIGHT violation
    end if
end for

function GETDIRECTION(A, i)
    return $i > 0 \wedge$ `cmp`($A[i-1], A[i]$) > 0 ? LEFT : RIGHT
end function
function FIXLEFTVIOLATION(A, i, `leftBound`)
    $t \leftarrow$ BINARYSEARCHLEFT($A, A[i],$ `leftBound`$, i-1$)
    MOVE(A, i, t)
    return t
end function
function FIXRIGHTVIOLATION(A, i, `rightBound`)
    $t \leftarrow$ BINARYSEARCHRIGHT($A, A[i], i+1,$ `rightBound`)
    MOVE(A, i, t);
    return t
end function

4.4 Correctness Proof

Lemma 5 (Violation Fix Invariant)**.** *Each fix operation during Phase 2 resolves a violation without introducing new ones.*

Proof. We fix each violation using binary search. For binary search to find the correct insertion point, the search range must contain no violations.

- *LEFT fix at index i*: The search range $[leftBound, i-1]$ contains no LEFT violations because LEFTs are processed left-to-right, and no RIGHT violations because all pending RIGHTs are flushed before any LEFT is fixed.
- *RIGHT fix at index i*: The search range $[i+1, rightBound]$ contains no RIGHT violations because RIGHTs are processed in LIFO order with $rightBound$ narrowing after each fix, and no LEFT violations because $rightBound$ never extends past the first LEFT in the segment.

□

Theorem 6 (Correctness)**.** *DeltaSort produces a correctly sorted array.*

Proof. The only violations in the array after Phase 1 are at updated indices. Phase 2 processes each updated index exactly once. By Lemma 5, each fix resolves a violation without introducing new ones. After all fixes, no violations remain, so the array is sorted. □

4.5 Complexity Analysis

Theorem 7 (Time Complexity)**.** *DeltaSort runs in $O(k \log k + k \log n + M)$ time, where M is the total movement.*

Proof. We analyze the two phases separately.

Phase 1. Sorting the k updated indices and their corresponding values costs $O(k \log k)$ time. Writing the sorted values back requires $O(k)$ time.

Phase 2. Each updated index is processed once. Violation checks cost $O(1)$ per index, and each repair performs a binary search over a sorted region in $O(\log n)$ time. Thus Phase 2 requires $O(k \log n)$ time. Let M denote the total number of values moved during repair. The movement cost is $O(M)$.

Total. The overall complexity is $O(k \log k + k \log n + M)$. □

Theorem 8 (Space Complexity)**.** *DeltaSort uses $O(k)$ auxiliary space.*

Proof. Phase 1 stores k updated indices and k updated values. Phase 2 maintains a pending stack of at most k indices. No $O(n)$ auxiliary structures are required. □

Remark 9 (Movement Efficiency)*.* While worst-case movement is $O(kn)$, the segmentation created by Phase 1 tends to reduce movement in the average case. Empirical results in §5 demonstrate substantial speedups, validating that segmentation (Lemma 4) effectively reduces movement in practice. We will analyze average-case movement more rigorously in future work.

Table 1 compares algorithm complexity with the standard baseline approaches used in the experiments:

- **NativeSort (NS)**: Re-sort the array using the natively available sort function. $O(n \log n)$ comparisons and movements.
- **Binary-Insertion-Sort (BIS)**: Extract updated values, then for each: binary search for correct position, reinsert. $O(k \log n)$ comparisons, $O(kn)$ worst-case movement. Searches the full array range for each insertion.
- **Extract-Sort-Merge (ESM)**: Extract updated values, sort them, merge with clean values. $O(k \log k + n)$ comparisons, $O(n)$ movement. Requires $O(n)$ auxiliary space.

Table 1: Algorithm complexity comparison

Algorithm	Comparisons	Movement	Space
NativeSort	$O(n \log n)$	$O(n \log n)$	$O(n)$
Binary-Insertion-Sort	$O(k \log n)$	$O(kn)$	$O(1)$
Extract-Sort-Merge	$O(k \log k + n)$	$O(n)$	$O(n)$
DeltaSort	$O(k \log n)$	$O(kn)$*	$O(k)$

*Worst case; average case is empirically much smaller.

5 Experimental Evaluation

All experiments are run using a Rust implementation of DeltaSort [5] on a synthetic dataset of user objects with composite keys (country, age, name) on an M3 Pro MacBook Pro with 18GB RAM. DeltaSort is compared against three baselines: NativeSort (Rust's `sort_by`), Binary-Insertion-Sort (BIS), and Extract-Sort-Merge (ESM).

5.1 Correctness

Correctness is formally proven in Theorem 6 and also verified by an extensive set of randomized unit tests [5] across various scales and update sizes. The test routine generates a sorted base array of size n, applies k random updates at random indices, runs DeltaSort, and asserts that the final array is sorted and contains all original values with updated values. All tests pass successfully.

5.2 Execution Time

Figure 2 shows execution time (in microseconds) for $n = 50$K values as a function of updated count k. DeltaSort consistently outperforms all alternatives up to approximately $k = 16.5$K (crossover point), achieving significant speedups of 4–20× over NativeSort in the intermediate range. Also note the orders-of-magnitude gap between DeltaSort and the baseline incremental algorithms (BIS and ESM) across the full range of k values.

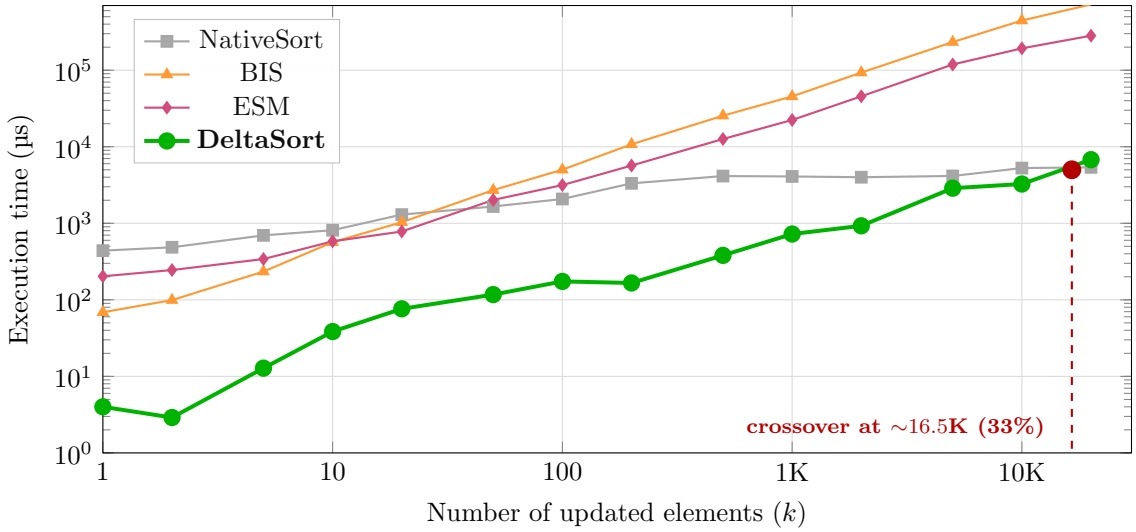

Figure 2: Execution time comparison for $n = 50$K elements (log-log scale). The vertical dashed line marks the crossover point ($k \approx 16.5$K) where DeltaSort's advantage over NativeSort diminishes.

5.3 Comparator Invocation Count

Figure 3 shows the number of comparator invocations for each algorithm. DeltaSort and BIS both achieve $O(k \log n)$ comparisons, substantially fewer than NativeSort's $O(n \log n)$ and ESM's $O(k \log k + n)$. The comparison counts for DeltaSort are 10–40% higher than BIS, indicating that DeltaSort's performance advantage comes from reduced data movement (Lemma 4). This also suggests that in scenarios with expensive comparators, the speedups from *DeltaSort* over NativeSort will be even greater.

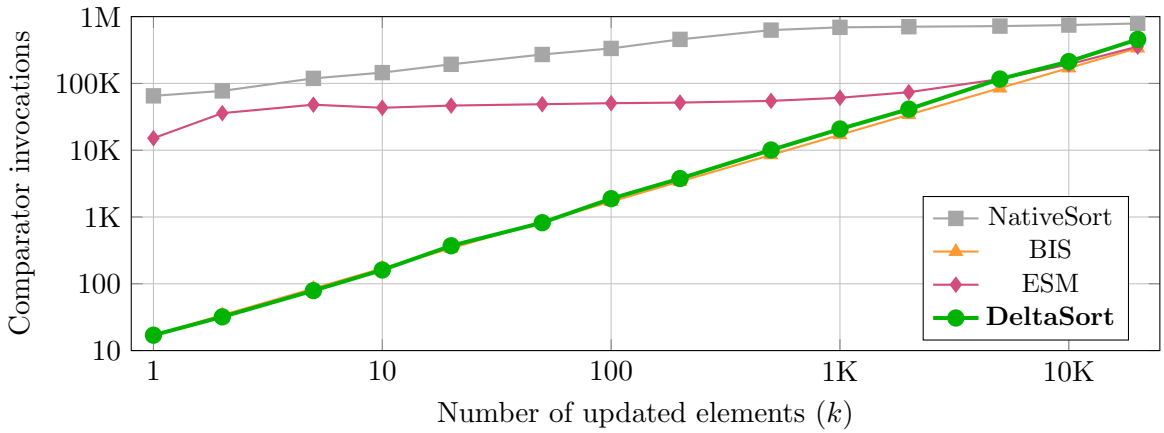

Figure 3: Comparator invocation count for $n = 50$K elements. DeltaSort and BIS both achieve $O(k \log n)$ comparisons, while NativeSort uses $O(n \log n)$ regardless of k, and ESM uses $O(k \log k + n)$. The 10–40% higher comparison counts for DeltaSort vs BIS confirm that movement confinement (Lemma 4) is the major factor in DeltaSort speedup.

5.4 Crossover Threshold Analysis

A key practical question is: at what delta size should one switch from DeltaSort to NativeSort? A binary search was conducted for the crossover point k_c across array sizes from 1K to 10M values.

Figure 4 visualizes how the crossover ratio k_c/n varies with array size. The ratio peaks around ∼33% for medium-sized arrays ($n \approx 50$K) and declines rapidly for very large arrays ($n > 500$K), suggesting that DeltaSort's advantage narrows as arrays grow very large. This could be because the segmentation does not grow linearly with array size and hence advantages diminish at larger scales. More study is needed to understand this trend fully.

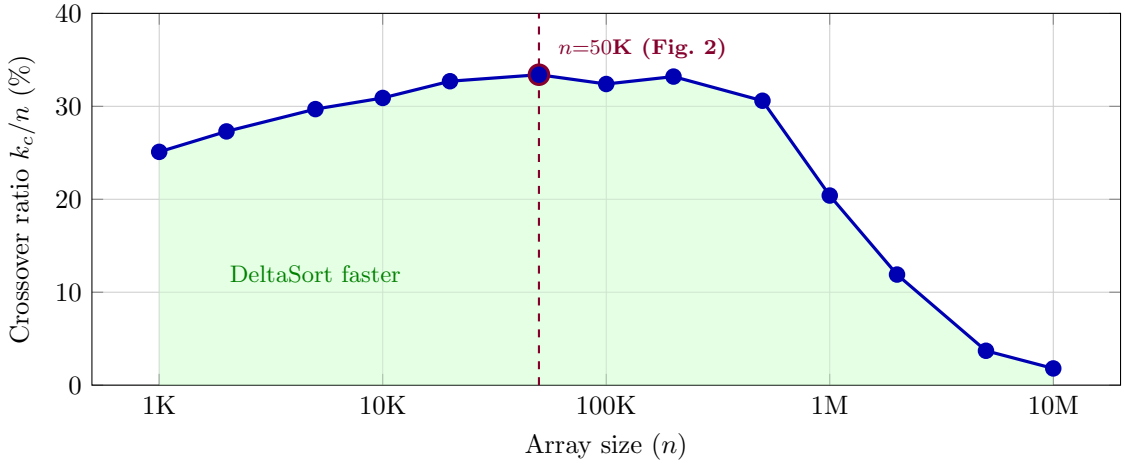

Figure 4: Crossover ratio k_c/n as a function of array size. DeltaSort outperforms NativeSort when the updated fraction is below the curve (shaded region).

The key takeaway is that DeltaSort offers the best incremental sort performance for **large** ranges of update sizes (0–30% for the dataset we tested). The exact crossover threshold depends on the specific scenario (array size, data types, comparator cost, etc.).

5.5 Performance in managed execution environments

DeltaSort was also implemented in JavaScript [5] and benchmarked on the V8 engine to evaluate behavior in managed runtimes. Initial results indicate that DeltaSort outperforms NativeSort for some workloads, but the gains are more modest and the crossover point occurs at smaller update sizes. This behavior appears to be driven primarily by the performance characteristics of the native JavaScript sort, which is highly optimized and handles nearly sorted inputs exceptionally well. As a result, the relative advantage of segmented repair is reduced in this environment compared to the Rust implementation. The JavaScript benchmarks are still being refined and will be reported in a later revision. Until then, the Rust implementation provides the primary and authoritative performance characterization.

6 Future Work

This work opens up several directions for future investigation:

- **Establish average-case movement bounds:** We established the efficiency gains from segmentation empirically, but the theoretical worst-case is still $O(nk)$. We need to establish average-case bounds for data movement under typical update distributions. It will also help explain observed crossover behavior at large scales and clarify when segmented repair is most effective.
- **Analyze structured workloads:** The current evaluation tests randomized updates, whereas many real workloads exhibit additional structure, such as gradual value changes in leaderboards or localized updates in interactive list views. Studying such patterns may reveal regimes where DeltaSort's advantages are amplified or diminished.
- **Study managed environments:** Performance variance in managed environments warrants deeper investigation. Understanding the impact of factors such as garbage collection and JIT compilation will help explain the observed performance characteristics.
- **Analyze block-structured storage:** Although this work focuses on in-memory arrays, the update-aware model naturally extends to block-structured storage. Exploring how DeltaSort-style segmentation interacts with page- or block-based layouts may clarify its applicability to database and external-memory settings.

7 Conclusion

This paper introduced *DeltaSort*, an incremental repair algorithm for maintaining sorted arrays. The key insight is that pre-sorting updated values induces segmentation: updated values naturally partition into segments that can be repaired independently. DeltaSort leverages this segmentation through stack-based processing. LEFT-moving values are repaired immediately with progressively narrowed search ranges, while RIGHT-moving values are deferred and processed in reverse order to ensure stable target positions. This segmentation avoids redundant comparisons and overlapping value movement that arise from repeated binary insertion. An experimental evaluation in Rust demonstrates that DeltaSort outperforms both blind native sorting and repeated binary insertion across a wide range of array sizes and update volumes.

More broadly, this work highlights the value of integrating application-level update information into core algorithms. When sorting routines are informed of which values changed, batching and segmentation become possible, enabling performance improvements that blind algorithms cannot realize. DeltaSort illustrates how modest structural insight—segmentation combined with disciplined processing order—can yield substantial practical gains.

References

[1] Georgy M. Adelson-Velsky and Evgenii M. Landis. An algorithm for the organization of information. *Proceedings of the USSR Academy of Sciences*, 146:263–266, 1962.

[2] Tyler Akidau et al. The dataflow model: A practical approach to balancing correctness, latency, and cost. In *VLDB*, 2015.

[3] Rudolf Bayer and Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173–189, 1972.

[4] Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $o(n \log n)$. In *Proceedings of the 3rd International Conference on Fun with Algorithms*, pages 16–23, 2004.

[5] Shubham Dwivedi. DeltaSort: Reference implementation and benchmarks. `https://github.com/shudv/deltasort`, 2026. Rust and JavaScript implementations with unit tests and benchmark suite.

[6] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, pages 8–21, 1978.

[7] Ashish Gupta and Inderpal Singh Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 1995.

[8] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, 2nd edition, 1998.

[9] Heikki Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, 100(4):318–325, 1985.

[10] David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997.

[11] Miloš Nikolić et al. Incremental computation with dataflow graphs. In *OOPSLA*, 2014.

[12] Orson R. L. Peters. Pattern-defeating quicksort. `https://github.com/orlp/pdqsort`, 2021.

[13] Tim Peters. Timsort. Python source code, 2002. Adopted by Java SE 7, Android, and V8.

[14] William Pugh. Skip lists: A probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.