DeltaSort: Incremental Sorting of Arrays with Known Updates

Shubham Dwivedi
Independent Researcher, Kanpur, India

Abstract

When records need to be read in a particular order, sorting at query time incurs repeated $\Theta(n \log n)$ cost for an array of n records and can become a bottleneck in read-heavy workloads. A common solution is to maintain a derived sorted read-replica that is kept updated as the underlying system-of-record changes. For updating read-replicas that are stored as arrays, existing approaches rely on either full re-sorting or incremental algorithms such as binary insertion or merge-based sort. In this paper, we study incremental sorting under a new model in which the sorting routine is explicitly informed of the k indices updated since the previous sort – a setting that naturally arises in systems that track updates. Under this model, we present *DeltaSort*, a new algorithm that runs in $O(n\sqrt{k})$ expected time using $O(k)$ auxiliary space under a random update model, and *outperforms existing algorithms for small update batches* in our experimental evaluation.

2012 ACM Subject Classification Theory of computation → Sorting and searching; Theory of computation → Data structures design and analysis

Keywords and phrases Incremental sorting, Sorting algorithms, Array maintenance

Digital Object Identifier 10.4230/LIPIcs.SEA.2026.18

Supplementary Material *Software (Source Code)*: `https://github.com/shudv/deltasort` [13] archived at `swh:1:dir:bf69484c0b9a682ec1aac98197b5df0a554ed4a3`

1 Introduction

Sorting is among the most heavily optimized primitives in modern systems, backed by decades of deep research. Standard library implementations – TimSort [4], Introsort [26], and DriftSort [8] deliver excellent performance for general inputs by exploiting existing 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 been updated since the previous sort. While this model is more general and makes things straightforward for the caller, it also *misses opportunities for optimization when such information is available*. Modern systems often maintain metadata about updated records as part of their execution pipeline [22], making it possible to expose this information to lower-level sorting primitives. In the absence of a primitive that can accept this as a first-class input, they must either perform a full re-sort or use standard insertion-based or merge-based incremental algorithms.

As a concrete example, consider a client application displaying a large dynamic sorted list (e.g., a task manager or interactive dashboard). Such applications have tight latency budgets [16, 10], and UI frameworks encourage memoization to avoid recomputing expensive derived values [29]. However, memoization is typically coarse-grained: any change invalidates the cache and forces a full re-sort [16, 15]. An update-aware sorting primitive could restore order incrementally, avoiding this overhead.

Once the indices of the updates are known, how do we exploit this extra information to restore order more efficiently than performing a full re-sort? While similar questions arise for a variety of ordered data structures (e.g., B-Trees, heaps, and BSTs), in this paper we deliberately focus on *arrays*, which provide a contiguous memory layout. This choice

Table 1 Algorithms for update-aware sorting.

Algorithm	Time (Expected / Worst)	Space
Binary-Insertion-Sort†	$O(nk)$	$O(1)$
Extract–Sort–Merge*†	$O(k \log k + n)$	$O(k)$
DeltaSort*	$O(n\sqrt{k})$ / $O(nk)$	$O(k)$

n = array size, k = number of updates

*These use a sort primitive internally, assumed to run in $O(k \log k)$ time.

†These have the same expected and worst-case time complexity.

is intentional: arrays remain the dominant representation for sorted data in performance-critical systems because they offer superior cache locality and enable SIMD/vectorized processing [20] – benefits that pointer-based structures fundamentally cannot provide. As a result, many systems prefer to preserve array layouts even when updates are frequent, rather than switching to fully dynamic structures [12, 1, 9]. This also allows us to cleanly isolate the algorithmic consequences of exposing update indices to the sorting routine and study the resulting trade-offs in a concrete setting.

Given an array of size n, with k updates, existing algorithms fall into two broad categories:

1. **Insertion-based**: Each updated value is inserted into its correct position one by one. We use Binary-Insertion-Sort (*BIS*) as a representative of this category, which uses binary search to find the correct position for each updated value. This approach uses only $O(1)$ auxiliary space but incurs $\Theta(nk)$ data movement, making it suitable only when $k \ll n$.
2. **Merge-based**: Updated values are extracted into a separate buffer, sorted using an efficient $O(k \log k)$ algorithm, and then merged back into the original array. We use Extract–Sort–Merge (*ESM*) as a representative of this category, which compacts the unchanged elements and merges the updated values from the back. It runs in $O(k \log k + n)$ time using only $O(k)$ space, but always touches the whole array irrespective of k.

The optimal strategy is therefore: use BIS when $k \ll n$, ESM otherwise. This raises the question: *are there algorithms that can outperform BIS and ESM* for practically relevant workloads? In this paper, we introduce such an algorithm. Specifically, this paper makes the following contributions:

1. **Update-aware sorting model:** We formulate a sorting model in which the sorting routine is explicitly informed of the updated indices since the previous sort. Under this model, BIS and ESM are the best known baseline algorithms[1] that naturally exploit knowledge of which indices have been updated.
2. **DeltaSort algorithm:** We present *DeltaSort*, an update-aware insertion-based sorting algorithm that reduces data movement by identifying *segments* – independent regions of the array – and restricting movement within them.
3. **Theoretical analysis:** Under a random bounded-range update model, we show that DeltaSort achieves $O(n\sqrt{k})$ expected time complexity using $O(k)$ auxiliary space.

[1] Other potential baselines either reduce to trivial combinations of BIS and ESM, or rely on in-place merge algorithms [24] that do not naturally fit our problem model. We evaluated multiple baselines and picked the best-performing ones (see source repository [13] for benchmarks).

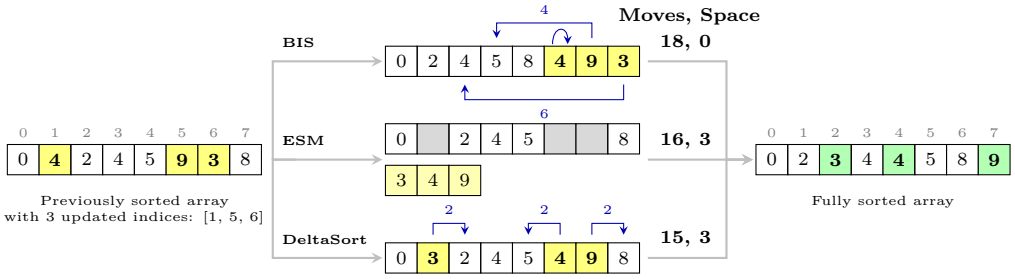

Figure 1 Movement and space comparison for sorting 3 updated values (yellow). In this example, ESM and DeltaSort sort updated values using insertion sort to simplify movement analysis. BIS first uses 8 moves to shift updated values to the end (since insertion requires a fully sorted prefix), then $0 + 4 + 6$ moves to insert each updated value into the sorted prefix. ESM uses $3 + 3$ moves to extract and sort the updated values, 4 moves for compacting the original array and then 6 moves for backward merge. DeltaSort uses $3+3+3$ moves to collect, sort, and write the updated values back to the array, then just $2+2+2$ moves to place each value correctly within its *segment* (see Definition 3), *without having to form a fully sorted prefix upfront* like BIS.

4. **Experimental validation:** We benchmark Rust implementations of DeltaSort, BIS, and ESM on synthetic datasets. In our evaluation, DeltaSort is the *fastest algorithm for small update batches* ($k \lesssim 1\%$ for $n = 100K$), which is the most practically relevant range for incremental workloads.

2 Related Work

Adaptive sorting algorithms exploit existing order in the input to improve performance on nearly sorted data. TimSort [4], DriftSort [8], and natural mergesort [23] 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 [25]. These approaches, however, *assume no explicit knowledge of which values have been updated.*

Incremental view maintenance (IVM) is a fundamental and deeply researched problem, where updates to input data are propagated to derived results using explicit delta representations [19, 30, 3, 5]. These techniques focus on maintaining query results, indexes, aggregates, and materialized views, and operate at the higher level of relational or dataflow operators rather than array-based sorting primitives. They treat deltas as first-class citizens and demonstrate the practical value of update-aware computation. However, they *do not specifically address the lower-level problem of incrementally sorting arrays.*

Partial sorting is another important problem where sorted output is read incrementally to avoid sorting everything upfront [11, 27]. This is *a different model* that addresses efficient selection rather than maintaining a fully sorted array under updates at all times.

Dynamic data structures offer a different trade-off. Self-balancing trees such as AVL trees [2], red–black trees [18], B-trees [6], and skip lists [28] support efficient ordered updates with logarithmic cost, but they *do not preserve the contiguous layout and its associated advantages* [12, 1, 9]. A related idea is "library sort" [7], which accelerates repeated insertions by maintaining gaps and hence *does not preserve strictly contiguous layout* by design.

In-place merging algorithms focus on merging two sorted arrays without using auxiliary space [24, 21]. While there is some conceptual overlap with DeltaSort in the nature of implementation, both the problem model and the goal differ. In-place merging assumes

both input arrays are already sorted and aims for $O(1)$ space usage, whereas DeltaSort deals with a sorted array where some values have been updated and can be arbitrarily interspersed within the array, and the aim is to offer a practical alternative to existing algorithms.

Overall, this work focuses on a more specialized (but fundamental) problem: maintaining sorted order in contiguous arrays after a batch of in-place updates, under the assumption that the indices of updated values are explicitly available. This setting is not addressed by adaptive sorting algorithms, system-level incremental indexing techniques, selection-based partial sorting algorithms, or in-place merging algorithms. DeltaSort operates as a low-level sorting primitive that complements higher-level incremental systems while preserving the cache locality of array-based layouts. To our knowledge, this sorting model has not been previously studied.

3 Problem Model

▶ **Definition 1** (Update-Aware Sorting). *Let $A[0..n-1]$ be an array of size n that is initially sorted according to a strict weak ordering induced by a comparator* `cmp`*. Let $U \subseteq \{0, \ldots, n-1\}$ be a set of k distinct indices such that the values at indices in U may have been arbitrarily updated*[2]*, while values at all other indices remain unchanged. The* update-aware sorting problem *is to restore the global sorted order of A with respect to* `cmp`*, given explicit knowledge of the update set U.*

The sorting primitive in this model requires the caller to store the k updated indices, incurring an extra $O(k)$ space overhead compared to full re-sorting. This overhead is modest and aligns with how updates are applied in practice: locating and updating a value already requires identifying its position, and recording that position incurs only constant additional cost per update. In contrast, avoiding this bookkeeping typically forces the system to fall back to either full re-sorting, which incurs $\Theta(n \log n)$ time cost per update, or BIS, which incurs $\Theta(n)$ data movement cost per update. Modern streaming and stateful processing systems [22] routinely track update deltas as part of their execution pipelines, making the update-aware model a natural and practically useful abstraction.

4 DeltaSort Algorithm

DeltaSort is an insertion-based update-aware sorting algorithm. Below, we first give an overview of the algorithm, then formally prove its correctness, and finally analyze its expected complexity under a random update model.

4.1 Overview

DeltaSort operates in two phases (see Algorithm 1 for pseudocode and Figure 2 for an example). The first phase establishes a structure among updated values, and the second phase exploits this structure to efficiently sort the array.

[2] This model can be easily extended to insertions and deletions as well. Insertions can be handled by appending the new value to the end of the array and adding n to U. Deletions can be reduced to updates by assigning a sentinel value larger than all existing values and truncating the array after sorting.

Algorithm 1 DeltaSort.

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

Phase 1: Establish structure
`updatedIndices` $\leftarrow$ sort(U)
`updatedValues` $\leftarrow$ sort($[A[u] : u \in$ `updatedIndices`$]$, `cmp`)
for $(u, v) \in$ zip(`updatedIndices`, `updatedValues`) **do**
    $A[u] \leftarrow v$
end for
`updatedIndices.push`(n) ▷ Sentinel for trailing R-segment

Phase 2: Fix violations
$\ell \leftarrow 0$; `pendingRight` $\leftarrow []$
for $i \in$ `updatedIndices` **do**
    `dir` $\leftarrow (i = n)$? LEFT : GETDIRECTION(A, i)
    if `dir` $=$ LEFT **then**
        $r \leftarrow i - 1$
        while `pendingRight` $\neq \emptyset$ **do**
            $j \leftarrow$ `pendingRight.pop`()
            if `cmp`$(A[j], A[j+1]) > 0$ **then**
                $r \leftarrow$ FIXRIGHT(A, j, r) $- 1$
            end if
        end while
        if $i < n$ **then** ▷ Skip the sentinel
            $\ell \leftarrow$ FIXLEFT(A, i, ℓ) $+ 1$
        end if
    else
        if `pendingRight` $= \emptyset$ **then** ▷ First R in a segment
            $\ell \leftarrow i$
        end if
        `pendingRight.push`(i) ▷ Defer R
    end if
end for

Phase 1: Establish structure

DeltaSort begins by extracting the values at the updated indices, sorting them (e.g., via indirect heapsort using the index list as indirection), and writing them back into the array in increasing index order. This step enforces order among updated indices: if $i < j$ are both updated indices, then $A[i] \leq A[j]$ (according to `cmp`) after Phase 1. The unchanged values are already in order by design. Hence, any remaining order violations must occur only between updated values and their unchanged neighbors.

Phase 2: Fix violations

With the above structure in place, DeltaSort fixes the array one updated index at a time left to right. When an updated index is first encountered, we first determine its *direction*:

▶ **Definition 2** (Direction). *During Phase 2, for an index $i \in U$, we define its* direction *based on its local order with its left neighbor:*

1. ***LEFT (L)**: Value is ordered incorrectly to its left neighbor and hence* must *move left: $i > 0$ and $A[i-1] > A[i]$*

Algorithm 2 Helper procedures.

```
1: function GetDirection(A, i)
2:     if i > 0 and A[i − 1] > A[i] then
3:         return Left
4:     else
5:         return Right
6:     end if
7: end function

8: function FixRight(A, j, r)
9:     p ← binary search for insertion position of A[j] in A[j+1..r]
10:    rotate A[j..p−1] left by one position          ▷ Shift A[j] to position p−1
11:    return p
12: end function

13: function FixLeft(A, i, ℓ)
14:    p ← binary search for insertion position of A[i] in A[ℓ..i−1]
15:    rotate A[p..i] right by one position           ▷ Shift A[i] to position p
16:    return p
17: end function
```

2. ***RIGHT (R)****: Value is ordered correctly to its left neighbor or it is the first value and hence* may *move right:* $i = 0$ *or* $A[i-1] \leq A[i]$

A few important things to note about this definition:

- Direction captures on which side of its current position the target of an updated index lies. Hence, direction is not defined (or needed) for unchanged indices ($i \notin U$).
- The definition of direction is asymmetric. L requires a strict violation ($A[i-1] > A[i]$), while R includes both violating and non-violating cases, as it may or may not violate order with its right neighbor. This is deliberate because DeltaSort treats all R values identically, so combining these cases simplifies the implementation (see Algorithm 1).

Phase 2 (lines 9–29) processes updated indices from left to right while maintaining a growing sorted prefix. We describe Phase 2 by first establishing a loop invariant and then proving its correctness by showing that the loop invariant holds at initialization, is preserved during maintenance, and that the array is fully sorted at termination.

State

We utilize the following state information during Phase 2:

- `updatedIndices`: Sorted list of updated indices with a sentinel n appended at the end.
- ℓ: The boundary of the sorted prefix, 0 at the start.
- `pendingRight`: Stack of pending R indices, empty at the start.

Loop invariant

At the start of each iteration of the loop at line 10, the following conditions hold for the current index i:

1. The subarray $A[0..\ell-1]$ is sorted.
2. There are no L violations in the range $[\ell, i-1]$.
3. The stack contains pending R indices within the range $[\ell, i-1]$.
4. If the previous updated index had L direction, then `pendingRight` is empty.

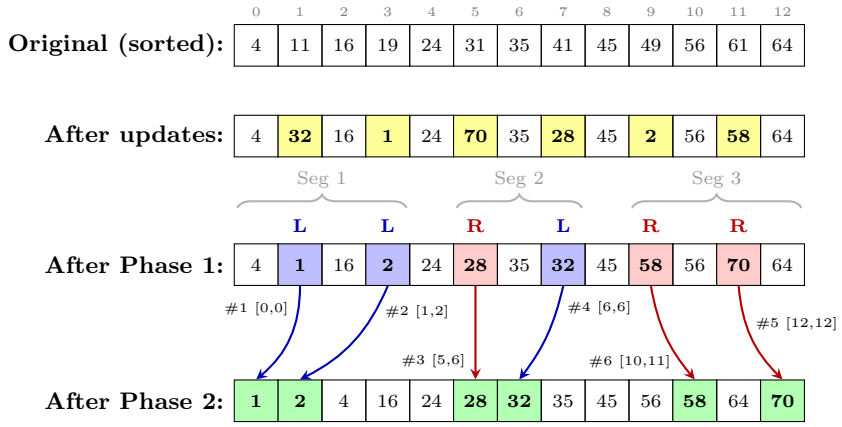

Figure 2 Segmentation example: An array of size 13 with 6 updates at indices 1, 3, 5, 7, 9, 11. In **Phase 1**, updated values are sorted among themselves and written back to the array. Each value is classified as L or R. In **Phase 2**, each value is fixed one by one in $2 + 3 + 0 + 2 + 2 + 2 = 11$ moves. The label numbers indicate the fixing order, along with the computed left and right bounds for binary search for each updated value.

Initialization

The invariant trivially holds before the first iteration ($i = \texttt{updatedIndices}[0]$).

Maintenance

We will now show that if the invariant holds for i, it also holds for the next updated index. Let `dir` denote the direction of i.

Case 1: `dir` $= L$.

1. We first set $r \leftarrow i - 1$. This marks the right boundary beyond which the R values in `pendingRight` cannot move, because after Phase 1, updated values are ordered and hence cannot cross each other in either direction.
2. We then flush all pending R indices j in LIFO order. Each R index j is fixed by binary insertion within the range $[j + 1, r]$, where r is advanced leftward after each fix to ensure that we search within a minimal range for maximum efficiency. By construction, the range $[j + 1, r]$ contains no violations, so binary search finds the correct insertion point. This step preserves conditions (3) and (4) of the loop invariant.
3. After all Rs are fixed, we fix the current L index i by binary insertion within $[\ell, i - 1]$. This range also contains no violations because all R violations are already fixed in the step above, and by invariant condition (2), this range contained no L violations to begin with. Finally, we set $\ell \leftarrow$ (insertion point of i) $+1$ (not needed for correctness, but reduces the search range). This preserves conditions (1) and (2) of the loop invariant.

Case 2: `dir` $= R$.

1. If `pendingRight` is empty, this is the first R in a new segment, so we set $\ell \leftarrow i$. This preserves conditions (1) and (2) of the loop invariant.
2. We then push i onto `pendingRight` for deferred fixing. This preserves conditions (3) and (4) of the loop invariant.

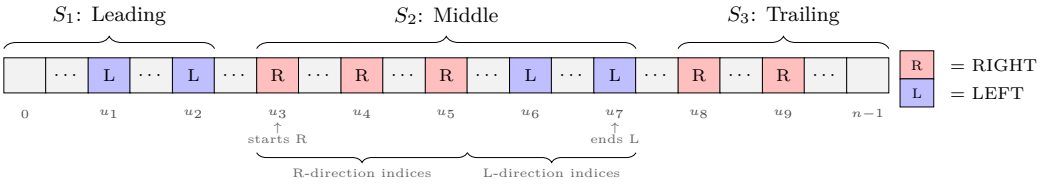

Figure 3 Segment structure (leading and trailing segments may be absent)

Termination

The loop processes all values of `updatedIndices`, including the sentinel index n, which is treated as L. Substituting $i = n$ in the invariant conditions, we have:

- Condition (1) implies that $A[0..\ell - 1]$ is sorted and contains its final values.
- Condition (2) implies there are no L violations in the range $[\ell, n - 1]$.
- Conditions (3) and (4) together imply that there are no R violations in the range $[\ell, n-1]$.
- Hence we can conclude that the array $A[0..n - 1]$ is fully sorted.

Figure 2 illustrates this sequence of operations for a sample array.

It is important to note that even though we are doing repeated binary insertions for fixing violations, the overall data movement is reduced as compared to naive independent binary insertions because the binary search range is smaller. To formalize this we need to understand the structure that emerges from Phase 1. If we picture the updated indices arranged in a sequence of L and R labels, based on their direction when first encountered during Phase 2, we can partition them into *segments*.

▶ **Definition 3** (Segment). *A* segment *is a subarray of A spanning indices $[i, j]$, where $0 \le i \le j < n$, such that:*

1. *Either $i = 0$, or $i \in U$ and has direction label R.*
2. *Either $j = n - 1$, or $j \in U$ and has direction label L.*
3. *There exist no updated indices $p, q \in U$ with $i < p < q < j$ such that p is L and q is R.*
4. *There exist no updated indices $p < i$ and $q > j$ such that (p, q) is also a segment.*

More intuitively, a segment is a maximal contiguous region of the array in which all R-direction updates precede all L-direction updates. The first two conditions define valid segment boundaries, the third enforces the R^*L^* structure within a segment, and the final condition ensures that segments are maximal and non-overlapping. Figure 3 illustrates this structure.

Key Insight: Segments can be fixed locally and independently

Unlike naive BIS, where each insertion can be at any point in the array, in DeltaSort, *an updated value in a segment cannot cross its segment boundary.* We formally prove this property next.

▶ **Lemma 4** (Movement Confinement). *During Phase 2, all value movement is confined within segment boundaries.*

Proof. Let S be a segment with R-indices $r_0, \ldots, r_{r-1}$ followed by L-indices $\ell_0, \ldots, \ell_{l-1}$, where $r + l \ge 1$. After Phase 1, updated values are ordered: $A[r_0] \le \cdots \le A[r_{r-1}] \le A[\ell_0] \le \cdots \le A[\ell_{l-1}]$. An R-value cannot have crossed ℓ_0 (if it exists), and an L-value cannot have crossed r_{r-1} (if it exists), since the above ordering must be preserved. Therefore, no value exits its segment. ◀

▶ Remark 5. Note that Lemma 4 also implies *segment independence*. Because no value exits its segment, segment lengths do not change. As a result, movement in a segment does not interfere with movement in another segment, which implies segments can be fixed independently in any order. This opens opportunities for **parallelization**, where segments could be fixed concurrently. We leave this investigation to future work.

4.2 Complexity Analysis

We analyze the expected total data movement incurred during Phase 2 of DeltaSort under a random bounded-range update model. In this model, updated values are drawn uniformly at random from a *fixed* value range. Choosing values at random avoids introducing any special structure that could bias the analysis.

▶ Remark 6 (Choice of Update Model). Several update models could be considered. For example, a *bounded-rank displacement* model constrains updates such that for every updated value $|Rank_{\text{new}} - Rank_{\text{old}}| \leq b$ for some bound b, while a *clustered update* model restricts updated indices to a region with $u_{\max} - u_{\min} \leq c$. Different models interact with DeltaSort's structure in different ways: bounded-rank displacement directly limits movement and is therefore favorable, whereas clustered updates may either help or hinder performance depending on the induced directional pattern. In this work, we adopt the random bounded-range update model as a *neutral* baseline. Specifically, updated indices are chosen uniformly at random from $\{0, \ldots, n-1\}$, and updated values are drawn independently from a fixed range of permissible values. This model introduces no additional structure that the algorithm can exploit, nor does it adversarially bias updates toward worst-case configurations (see Figure 4 for an illustration). Analyzing DeltaSort under more structured update models is left to future work.

▶ **Lemma 7** (Expected Movement). *Under the random bounded-range update model, for an array of size n with k updated indices, the expected total data movement incurred during DeltaSort's Phase 2 is $O(n\sqrt{k})$.*

Proof. After Phase 1, we have two sorted sequences of k values each:

- **Indices:** The k updated positions $i_0 < i_1 < \cdots < i_{k-1}$, sorted.
- **Values:** The k updated values $v_0 \leq v_1 \leq \cdots \leq v_{k-1}$, sorted.

Phase 1 pairs these by rank: the j-th smallest value v_j is placed at the j-th smallest index i_j. Since both indices and values are drawn uniformly at random, each sorted sequence forms an *order statistic* of k uniform samples.

Although values may be drawn from a different range than indices, what matters for determining direction is the value's *target position* – where it belongs in the final sorted array. For a value v_j in a sorted array of n values spanning range $[0, M)$, its target position is approximately $t_j = v_j \cdot n/M$. Since v_j is uniform over $[0, M)$, the target position t_j is uniform over $[0, n)$ – the same range as the indices. Thus, we can analyze both sequences as order statistics of uniform samples from the same range.

Define the *gap* at rank j as:

$$g_j = t_j - i_j$$

This measures how far ahead (positive) or behind (negative) the value's target position is relative to its current position. By Definition 2:

- $g_j < 0$ implies the value is too small for its position, so direction is L.
- $g_j \geq 0$ implies the value is adequate or too large, so direction is R.

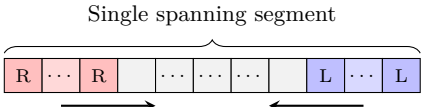

Figure 4 Worst-case configuration for DeltaSort is formed when updated values cluster at ends of the array and form a single spanning segment.

The gap evolves as we move through the ranks. Let $\Delta i_j = i_{j+1} - i_j$ and $\Delta t_j = t_{j+1} - t_j$ denote the spacings between consecutive order statistics. The change in gap is:

$$g_{j+1} - g_j = \Delta t_j - \Delta i_j$$

Since both Δt_j and Δi_j are spacings of independent uniform order statistics, they have the same expected value $(n/(k+1))$ and are independent of each other. Therefore, the increment $g_{j+1} - g_j$ has mean zero and behaves as a random step. This means the sequence $g_1, g_2, \ldots, g_k$ evolves as a *symmetric random walk*.

By Definition 3, segment boundaries occur at $L \to R$ transitions – precisely when the gap g_j crosses from negative to non-negative. This corresponds to an *upcrossing* of zero in the random walk. A classical result from random walk theory [14] states that a symmetric random walk of k steps has expected number of zero crossings $\Theta(\sqrt{k})$. Since $L \to R$ transitions account for roughly half of all zero crossings, the expected number of segments is $O(\sqrt{k})$.

With $O(\sqrt{k})$ segments partitioning the array of size n, the expected width of each segment is $O(n/\sqrt{k})$. By Lemma 4, movement is confined within segments. Since each of the k updated values moves at most the width of its containing segment, the expected total movement is:

$$\mathbb{E}[\text{total movement}] \leq k \cdot O\left(\frac{n}{\sqrt{k}}\right) = O(n\sqrt{k}) \qquad \blacktriangleleft$$

▶ Remark 8 (Worst Case Movement). While the expected movement is $O(n\sqrt{k})$, the worst-case movement is $O(nk)$. This occurs when updated values form a single segment spanning the entire array. This happens when updates *cluster monotonically* at the start or end of the array (see Figure 4). Hence, in a practical setting, a hybrid algorithm will need to fall back to ESM or full re-sort based on the number of segments.

▶ **Lemma 9** (Comparison Count). *DeltaSort performs $O(k \log n)$ comparisons.*

Proof. Phase 1 sorts the k updated values, requiring $O(k \log k)$ comparisons. In Phase 2, each updated value is fixed using binary search within its containing segment. As shown in the proof of Lemma 7, the expected number of segments is $O(\sqrt{k})$, so the expected width of a segment is $O(n/\sqrt{k})$. Since searches are confined within segment boundaries, each fix requires $O(\log(n/\sqrt{k}))$ comparisons, for a total of $O(k \log(n/\sqrt{k}))$ comparisons in Phase 2. Summing both gives:

$$O(k \log k) + O(k \log(n/\sqrt{k})) = O\big(k(\log k + \log n - \tfrac{1}{2} \log k)\big) = O(k \log(n\sqrt{k})) = O(k \log n) \blacktriangleleft$$

▶ **Theorem 10** (Time Complexity). *DeltaSort runs in $O(n\sqrt{k})$ expected time.*

Proof. By Lemma 7, the expected total data movement is $O(n\sqrt{k})$. By Lemma 9, the total number of comparisons is $O(k \log n)$. Phase 1 also contributes $O(k \log k)$ movement for sorting k values. Since $k \log n = o(n\sqrt{k})$ for $k = o(n)$, the total is $O(n\sqrt{k})$. ◀

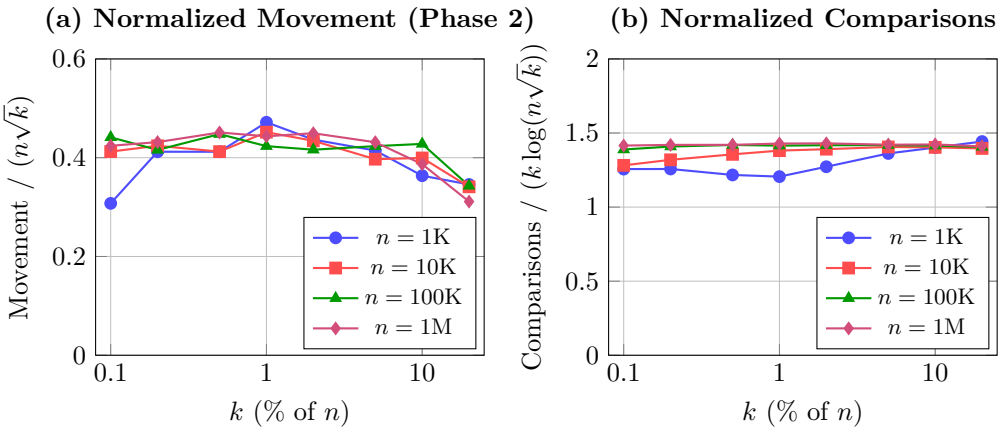

Figure 5 Empirical validation of DeltaSort's complexity bounds. Each line represents a different array size n. (a) Movement normalized by $n\sqrt{k}$ converges to ≈ 0.4, confirming $O(n\sqrt{k})$. (b) Comparisons normalized by $k \log(n\sqrt{k})$ converge to ≈ 1.4, confirming $O(k \log n)$. Convergence across scales validates the asymptotic bounds.

▶ **Theorem 11** (Space Complexity). *DeltaSort uses $O(k)$ auxiliary space.*

Proof. Phase 1 sorts k values using $O(k)$ space. Phase 2 maintains a pending stack of at most k indices. Therefore, the total auxiliary space is $O(k)$. ◀

4.3 Empirical Validation

Figure 5 presents empirical validation of DeltaSort's complexity bounds across multiple scales of n and k. For movement complexity, we normalize the measured movement by $n\sqrt{k}$; for comparison complexity, we normalize by $k \log(n\sqrt{k})$. If the theoretical bounds are tight, these normalized values should be approximately constant across all scales.

The results confirm both bounds: movement normalized by $n\sqrt{k}$ converges to approximately 0.4 across four orders of magnitude of n, and comparisons normalized by $k \log(n\sqrt{k})$ converge to approximately 1.4. The overlap of curves for different n validates that the asymptotic analysis correctly captures the algorithm's behavior.

5 Experimental Evaluation

All benchmarks are run on Rust implementations of each algorithm on randomly generated synthetic datasets of user objects (name, age, country) on an Apple M3 Pro chip. We chose Rust due to its predictable performance characteristics[3]. We used a realistic multi-key comparator to simulate a practical workload. Execution times are measured after sufficient warm-up, and each data point is the mean over repeated runs with 95% CI within 5%. All source code, along with an extensive suite of randomized correctness tests, is available for reproducibility [13].

[3] In contrast, managed-runtime environments like JavaScript on V8 are affected by garbage collection, non-contiguous memory layouts [17], etc.

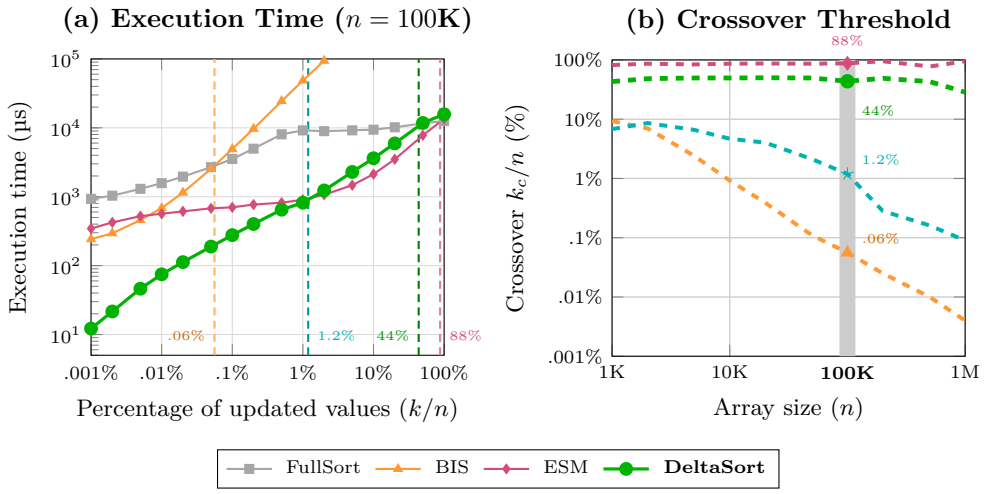

Figure 6 Rust benchmark. (a) Execution time for $n = 100$K on log–log scale. Dashed lines mark crossovers (BIS, DeltaSort, ESM vs. FullSort; DeltaSort vs. ESM in cyan). (b) Crossover threshold (k_c/n) across scales; the shaded line at $n = 100$K corresponds to (a). See Appendix B for raw data.

We evaluated the following algorithms:

- **FullSort**: Rust's `slice::sort_by` implementation based on DriftSort [8], representing full re-sorting of the array. This serves as a critical baseline to identify the crossover threshold k_c for each update-aware algorithm beyond which FullSort is preferable. We deliberately choose the adaptive stable sort over `sort_unstable_by` because DriftSort detects existing sorted runs, making it a *stronger* baseline for nearly-sorted inputs.
- **BIS** and **ESM**: Standard baseline algorithms as defined in Section 1. ESM uses Rust's unstable sort for sorting the updated values.
- **DeltaSort**: Standard implementation as described in Section 4. Sorts updated values using Rust's unstable sort, then fixes violations using binary search and rotation.

5.1 Results

Figure 6(a) shows execution time (in µs) for $n = 100$K values as a function of percentage of updated values on a log–log scale. We use a log–log scale to highlight interesting behavior at lower ranges of k, which is the most practically relevant range, since updates typically affect only a small fraction of an array at once. As k increases, all update-aware algorithms eventually lose to FullSort at their crossover threshold k_c, as the overhead of processing updates overshadows any benefit from knowing what was updated. Figure 6(b) shows how k_c varies for each algorithm across various scales. Several observations emerge:

1. The asymptotic behavior for each algorithm aligns with theory. BIS exhibits steep growth consistent with its $\Theta(nk)$ movement cost, quickly becoming impractical as k increases. ESM is relatively flat for small k, where its linear merge pass dominates. DeltaSort exhibits sublinear growth consistent with its $O(n\sqrt{k})$ expected movement.
2. **DeltaSort is the fastest algorithm for $k \lesssim 1\%$.** For example, at $k = 0.1\%$, DeltaSort is ∼13× faster than FullSort, ∼2.5× faster than ESM, and ∼18× faster than BIS. Even though 1% is small in absolute terms, it *aligns very well with practical workloads* where updates usually affect only a small percentage of the full dataset.
3. ESM is unsurprisingly the fastest algorithm across the intermediate range ($1\% \lesssim k \lesssim 88\%$), where its linear merge pass is most efficient.

4. Crossover thresholds for ESM and DeltaSort are high and largely consistent across the scales we tested, while BIS's crossover thresholds drop rapidly as array size increases. This shows that even though DeltaSort is insertion-based like BIS, it is able to leverage the segment structure to achieve high thresholds. On the other hand, DeltaSort–ESM crossover drops as array size increases, indicating that as array sizes get larger, *the range of k for which DeltaSort is the fastest algorithm narrows*.

These observations also suggest that, much like hybrid blind sorting algorithms (e.g., TimSort [4], DriftSort [8]), it would be beneficial to construct *adaptive update-aware* strategies. As an example, for the Rust implementation evaluated here, a simple adaptive strategy could be: use DeltaSort for $k \lesssim 1\%$, ESM for $1\% \lesssim k \lesssim 88\%$, and FullSort for $k \gtrsim 88\%$. The optimal crossover thresholds depend on factors such as update distribution and comparator cost. For example, as the comparator cost grows, DeltaSort preserves its advantage over BIS because both have similar comparison count: $O(k \log n)$, while widening its gap relative to ESM[4], which has much higher comparison overhead. Hence, the *crossover thresholds would shift in favor of DeltaSort*, expanding the range of k for which it is the fastest algorithm.

5.2 Performance in V8 runtime

DeltaSort was also implemented in JavaScript [13] and benchmarked on V8 against the native `Array.prototype.sort` to evaluate behavior in a runtime that *does not have a predictable data movement cost model* [17]. The key takeaways are (see Appendix A for full benchmarks):

1. DeltaSort performs similarly to BIS and has lower crossover thresholds than in Rust, indicating that **the asymptotic improvement seen in Rust is lost in V8**.
2. ESM is $\sim$1.5$\times$ faster than FullSort for k up to $\approx$50%, indicating that even in managed runtimes, exposing updated indices can unlock better incremental sort performance.

6 Conclusion

In this paper, we studied the problem of maintaining sorted arrays under incremental *known* updates. We argued that this *update-aware* setting arises naturally in many practical systems. Within this model, we presented *DeltaSort*, an incremental sorting algorithm that batches multiple updates instead of applying them independently. The key idea is to exploit structure created by pre-sorting updated values, which induces segments and enables localized, non-overlapping fixes. This enables DeltaSort to reduce redundant data movement. Our theoretical analysis shows that DeltaSort matches the comparison efficiency of BIS while exhibiting $O(n\sqrt{k})$ data movement under the random bounded-range update model. Experimental results in Rust demonstrate that DeltaSort outperforms both BIS and ESM for small update batches, which is the most practically relevant operating range for incremental workloads. For larger update batches, ESM's linear merge pass becomes preferable. To stress-test the underlying cost assumptions, we also evaluated DeltaSort in the V8 runtime, where data movement costs are less predictable. In this setting, DeltaSort and BIS perform similarly and exhibit much lower crossover thresholds than in Rust, highlighting that DeltaSort's full complexity advantage depends on execution environments with predictable and contiguous memory layouts.

[4] We also tested a binary-search-based variant of ESM that has lower comparison overhead but does not achieve the fastest execution time overall (see source repository [13]).

More broadly, this work suggests that when update metadata is already available, sorting primitives need not discover structure dynamically. By extending update-awareness down to low-level ordering primitives, we open a new space of algorithmic trade-offs beyond what blind sorting algorithms can achieve.

7 Future Work

While our analysis focuses on a random bounded-range update model, many practical workloads exhibit additional structure. Examples include *bounded-rank* or *clustered* updates. These models may improve or worsen the movement and comparison costs of DeltaSort, potentially impacting the range in which it is applicable. Hence, a more systematic study of such update models is important.

Additionally, we showed how segments can be fixed *locally* and *independently*. This strongly suggests opportunities for *parallel execution*, where different segments could be fixed concurrently without interference. In contrast, BIS is inherently sequential due to overlapping in-place shifts. Exploring the parallel variant of DeltaSort and understanding its scalability on multi-core systems is a promising direction for future work.

None of the incremental algorithms studied here (BIS, DeltaSort, ESM) guarantee *stability*. The impact of guaranteeing stability on the performance and complexity of update-aware algorithms remains to be studied.

Finally, while this paper treats DeltaSort as a standalone algorithm for analysis, practical systems would benefit from *adaptive update-aware strategies* that select among BIS, DeltaSort, ESM, and full re-sorting based on the update batch size and system constraints. Designing heuristic-based, low-overhead mechanisms for such dynamic selection is an open challenge.

References

1 Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. Column-stores vs. row-stores: how different are they really? In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 967–980, New York, NY, USA, 2008. Association for Computing Machinery. doi:10.1145/1376616.1376712.

2 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.

3 Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J. Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, and Sam Whittle. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proc. VLDB Endow.*, 8(12):1792–1803, August 2015. doi:10.14778/2824032.2824076.

4 Nicolas Auger, Vincent Jugé, Cyril Nicaud, and Carine Pivoteau. On the worst-case complexity of TimSort. In *Proceedings of the 26th Annual European Symposium on Algorithms (ESA)*, pages 4:1–4:13, 2018. doi:10.4230/LIPIcs.ESA.2018.4.

5 Ahmet Arif Aydin and Kenneth M. Anderson. Incremental sorting for Large Dynamic Data Sets. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pages 170–175, 2015. doi:10.1109/BigDataService.2015.35.

6 R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. Association for Computing Machinery. doi:10.1145/1734663.1734671.

7 Michael A. Bender, Martin Farach-Colton, and Miguel A. Mosteiro. Insertion sort is $o(n \log n)$. *CoRR*, cs.DS/0407003, 2004. doi:10.48550/arXiv.CS/0407003.

8 Lukas Bergdoll. driftsort: an efficient, generic and robust stable sort implementation. https://github.com/Voultapher/driftsort, 2023. Accessed: 2026-01-10.

9 Peter A. Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining query execution. In *2nd Biennial Conference on Innovative Data Systems Research (CIDR)*, 2005. URL: http://cidrdb.org/cidr2005/papers/P19.pdf.

10 Stuart K. Card, Thomas P. Moran, and Allen Newell. *The Psychology of Human-Computer Interaction*. Lawrence Erlbaum Associates, 1983.

11 John M. Chambers. Algorithm 410: Partial Sorting. *Communications of the ACM*, 14(5):357–358, 1971. doi:10.1145/362588.362591.

12 Ulrich Drepper. What every programmer should know about memory. https://people.freebsd.org/~lstewart/articles/cpumemory.pdf, 2007.

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

14 William Feller. *An Introduction to Probability Theory and Its Applications*, volume 1. Wiley, 3rd edition, 1968.

15 Google Chrome Team. The rail performance model. https://web.dev/rail/, 2015.

16 Google Chrome Team. Interaction to next paint (inp). https://web.dev/articles/inp, 2023. Accessed: 2026-01-10.

17 Google V8 Team. Elements kinds in V8. https://v8.dev/blog/elements-kinds, 2017.

18 Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science (sfcs 1978)*, pages 8–21, 1978. doi:10.1109/SFCS.1978.3.

19 Ashish Gupta and Inderpal Singh Mumick. *Maintenance of materialized views: problems, techniques, and applications*, pages 145–157. MIT Press, Cambridge, MA, USA, 1999.

20 John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 6 edition, 2017.

21 Bing-Chao Huang and Michael A. Langston. Practical in-place merging. *Commun. ACM*, 31(3):348–352, March 1988. doi:10.1145/42392.42403.

22 Martin Kleppmann. *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

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

24 M. A. Kronrod. Optimal ordering algorithm without operational field. *Soviet Mathematics Doklady*, 10:744–748, 1969.

25 Heikki Mannila. Measures of Presortedness and Optimal Sorting Algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, 1985. doi:10.1109/TC.1985.5009382.

26 David R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997. doi:10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-\#.

27 Rodrigo Paredes and Gonzalo Navarro. Optimal incremental sorting. In *2006 Proceedings of the Eighth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 171–182. SIAM, 2006. doi:10.1137/1.9781611972863.16.

28 William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, June 1990. doi:10.1145/78973.78977.

29 React Team. usememo. https://react.dev/reference/react/useMemo, 2024. Accessed: 2026-01-10.

30 Barbara G. Ryder and Marvin C. Paull. Incremental data-flow analysis algorithms. *ACM Trans. Program. Lang. Syst.*, 10(1):1–50, January 1988. doi:10.1145/42192.42193.

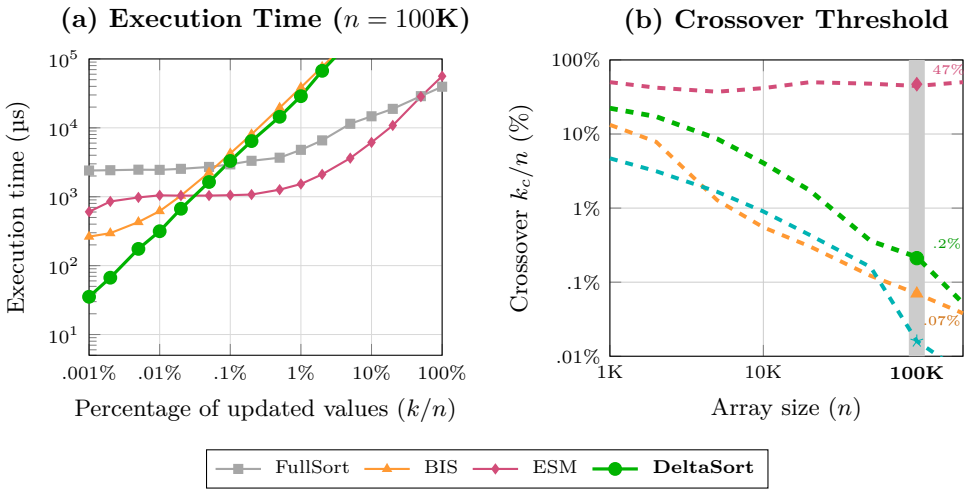

Figure 7 JavaScript/V8 benchmark results. (a) Execution time for $n = 100$K on log–log scale. (b) Crossover threshold (k_c/n) across scales; the shaded line at $n = 100$K corresponds to (a). Dashed lines in (b) show the crossover trends for each algorithm (BIS, DeltaSort, ESM vs. FullSort; DeltaSort vs. ESM in cyan).

A JavaScript/V8 Benchmarks

Both BIS and DeltaSort perform similarly and have rapidly declining crossover thresholds as n increases. This indicates that *V8's data movement cost model does not provide any asymptotic advantage for DeltaSort*. Hence, in V8, a valid adaptive strategy can be: use BIS/DeltaSort for $k \ll n$, ESM for $k \lesssim 50\%$, `Array.prototype.sort` for $k \gtrsim 50\%$.

B Raw Benchmark Data

Table 2 presents the raw execution time measurements for Rust benchmarks. Each row shows the mean execution time and 95% confidence interval as a percentage for a given value of k.

Table 2 Raw execution time data (Rust). $n = 100000$, machine: Apple M3 Pro, max CI: 5.00%, timestamp: 1775460239134. Times in μs shown as mean ± 95% CI%.

k	Iters	FullSort	±%	BIS	±%	ESM	±%	DeltaSort	±%
1	10,000	931	0.4%	243	0.3%	345	0.6%	12	3.3%
2	5,000	1,036	0.3%	295	0.4%	424	0.6%	22	3.2%
5	5,000	1,302	0.3%	458	0.8%	523	0.5%	46	1.9%
10	1,000	1,574	0.6%	684	0.8%	566	0.4%	75	3.2%
20	500	1,957	0.6%	1,149	0.9%	613	0.3%	112	3.6%
50	500	2,686	0.4%	2,542	0.6%	678	0.2%	189	3.4%
100	400	3,528	0.3%	4,917	0.6%	702	0.1%	277	3.8%
200	400	4,983	0.2%	9,689	0.4%	770	0.4%	402	3.4%
500	300	8,047	0.6%	24,466	0.3%	820	0.1%	649	3.9%
1,000	300	9,182	0.1%	48,576	0.2%	903	0.2%	819	3.8%
2,000	300	8,953	0.3%	94,043	0.3%	1,068	0.2%	1,234	3.3%
5,000	100	9,203	0.4%		0%	1,471	0.3%	2,289	5%
10,000	100	9,402	0.2%		0%	2,127	0.2%	3,618	3.5%
20,000	100	10,187	0.3%		0%	3,498	0.3%	5,940	2.5%
50,000	100	11,619	0.4%		0%	7,777	1%	11,765	1.4%
100,000	100	12,568	0.5%		0%	13,507	0.2%	15,745	0.4%