Matryoshka B+ Tree: Insert/Delete Performance Report

Comparative Benchmark Results

2026-02-20T07:49:45

Parameter	Value
CPU	13th Gen Intel(R) Core(TM) i7-1370P
L1d Cache	32 KB
L2 Cache	2 MB
L3 Cache	24 MB
Kernel	6.17.10-300.fc43.x86_64
Page Size	4096 B

Generated by `bench_compare` — matryoshka project

Contents

1 Introduction

This report evaluates the **matryoshka** B+ tree — a SIMD-blocked B+ tree using the FAST hierarchical layout (Kim et al., 2010) — against several tree and ordered-map libraries on *insert-heavy* and *delete-heavy* workloads. Goals:

1. Quantify the modification throughput gap across dataset sizes (65,536 to 16,777,216 keys).
2. Identify micro-architectural bottlenecks (cache misses, TLB pressure, branch misprediction) that explain the differences.

All measurements use `clock_gettime(CLOCK_MONOTONIC)`. Results are reported as Mop/s and ns/op.

2 Library Descriptions

Table 1: Libraries under test.

Name	Label	Description
`matryoshka`	Matryoshka B+ tree	SIMD-blocked B+ tree (FAST layout), 511-key 4 KiB leaves, arena a
`std_set`	std::set (RB tree)	Red-black tree (libstdc++), pointer-chasing, 40–48 B/node
`tlx_btree`	TLX btree_set	Cache-conscious B+ tree, sorted-array leaves ($B \approx 128$)
`libart`	libart (ART)	Adaptive Radix Tree, 4-byte keys, no predecessor search
`abseil_btree`	Abseil btree_set	Google B-tree, sorted-array leaves ($B \approx 256$)

3 Workload Descriptions

Table 2: Benchmark workloads.

Workload	Description
`seq_insert`	Insert N keys in ascending order. Exercises append paths.
`rand_insert`	Insert N unique keys in random order. Stresses leaf splits.
`ycsb_a`	95% insert / 5% search. Write-dominated OLTP model.
`rand_delete`	Bulk-load N sorted keys, delete all in random order.
`mixed`	Bulk-load N keys, then N alternating insert/delete ops.
`ycsb_b`	Bulk-load N keys, then 50% delete / 50% search.
`search_after_churn`	Bulk-load N keys, $N/2$ mixed churn (untimed), then 5,000,000 random predecessor searches.

4 Results: Insert-Heavy Workloads

4.1 Sequential Insert

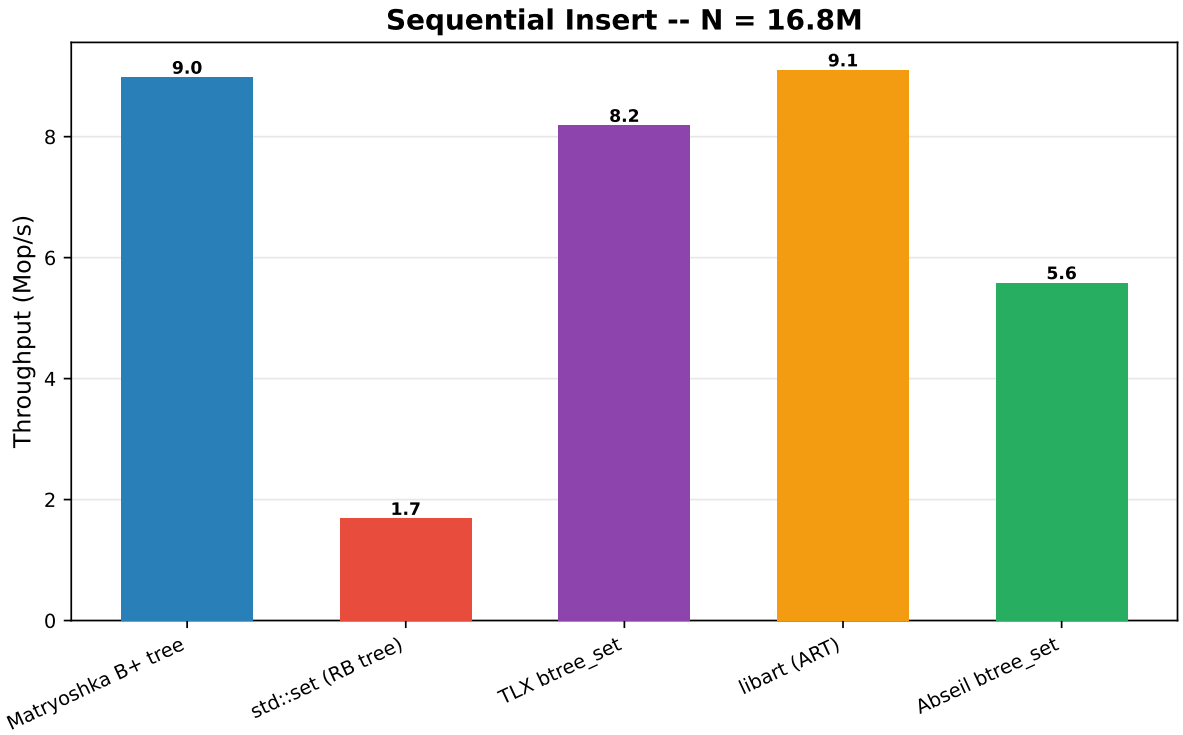

Figure 1: Sequential insert throughput (Mop/s).

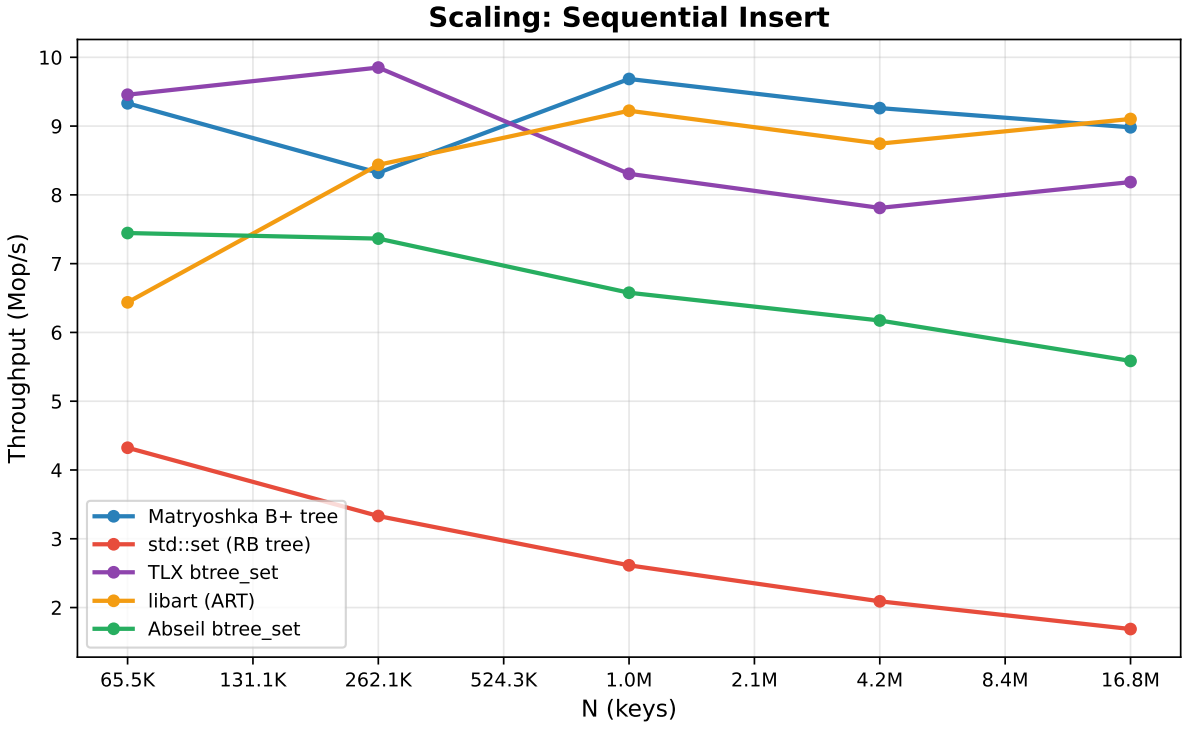

Figure 2: Sequential insert scaling.

4.2 Random Insert

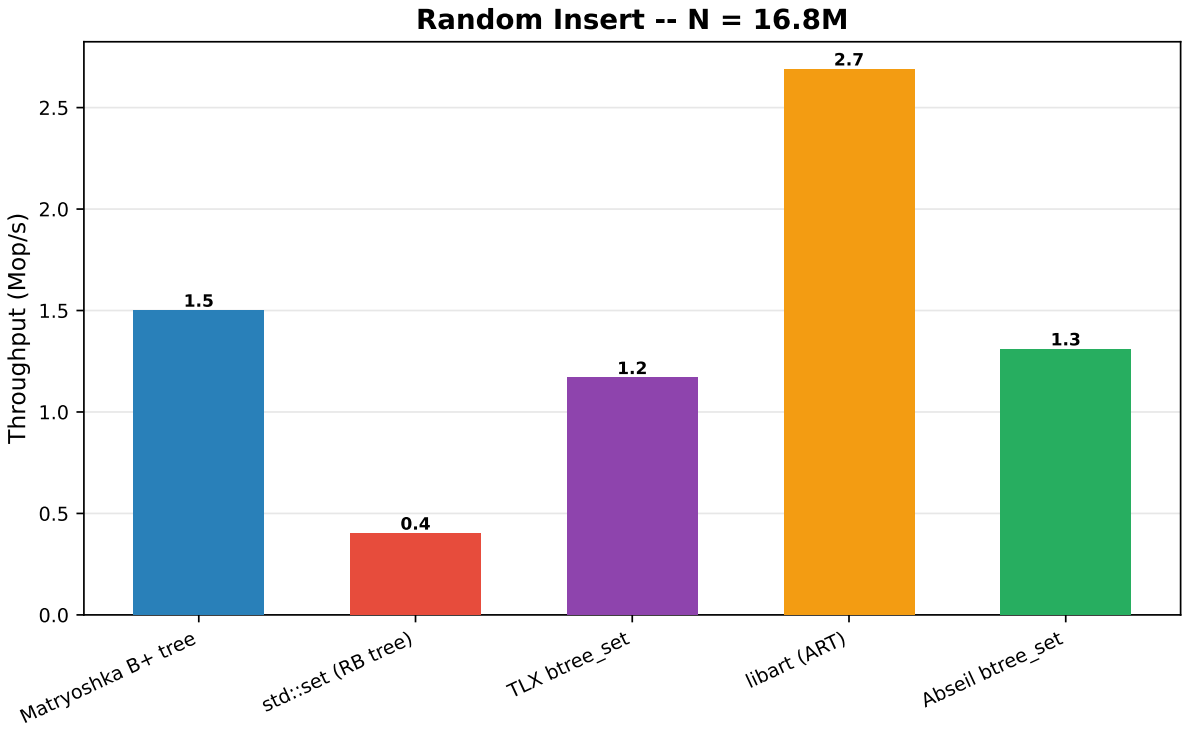

Figure 3: Random insert throughput (Mop/s).

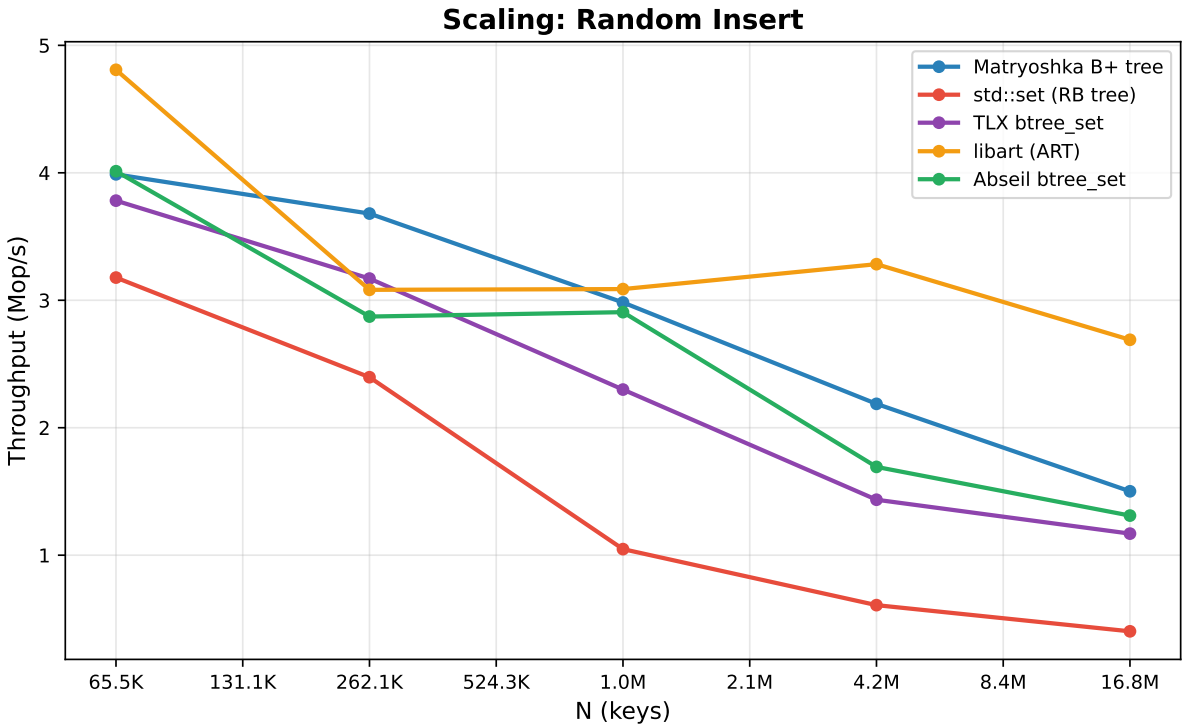

Figure 4: Random insert scaling.

4.3 YCSB-A (95% Insert / 5% Search)

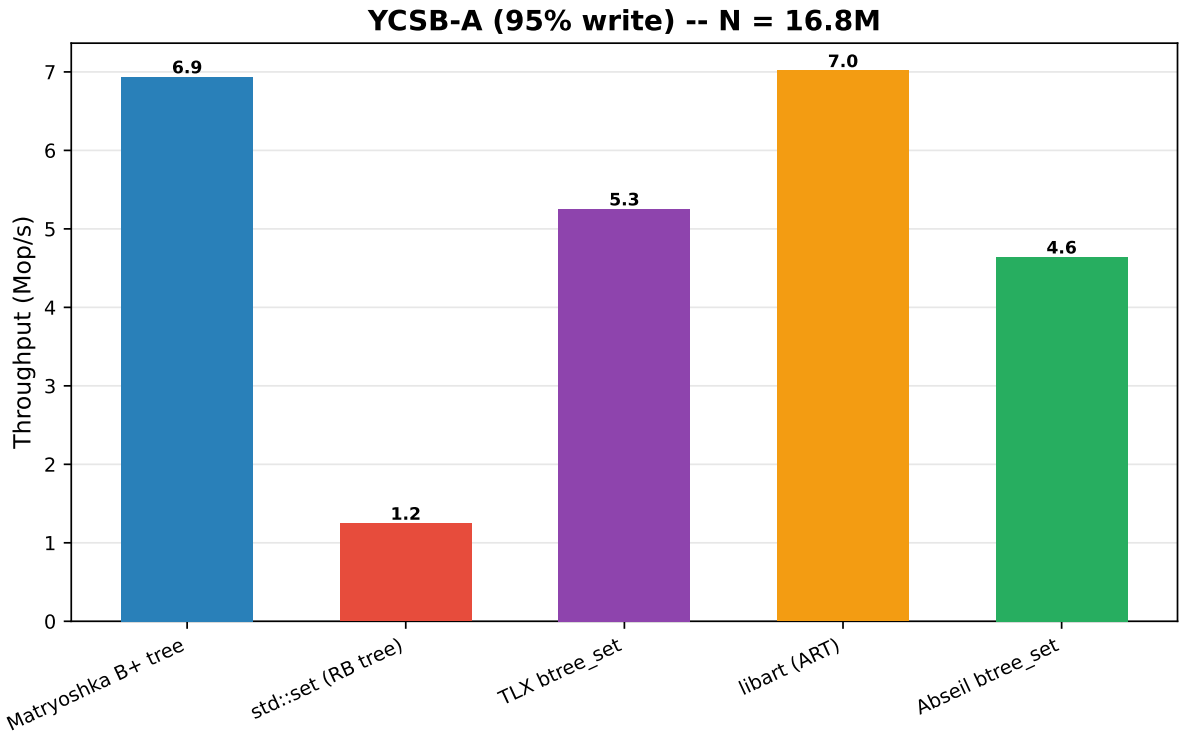

Figure 5: YCSB-A throughput (Mop/s).

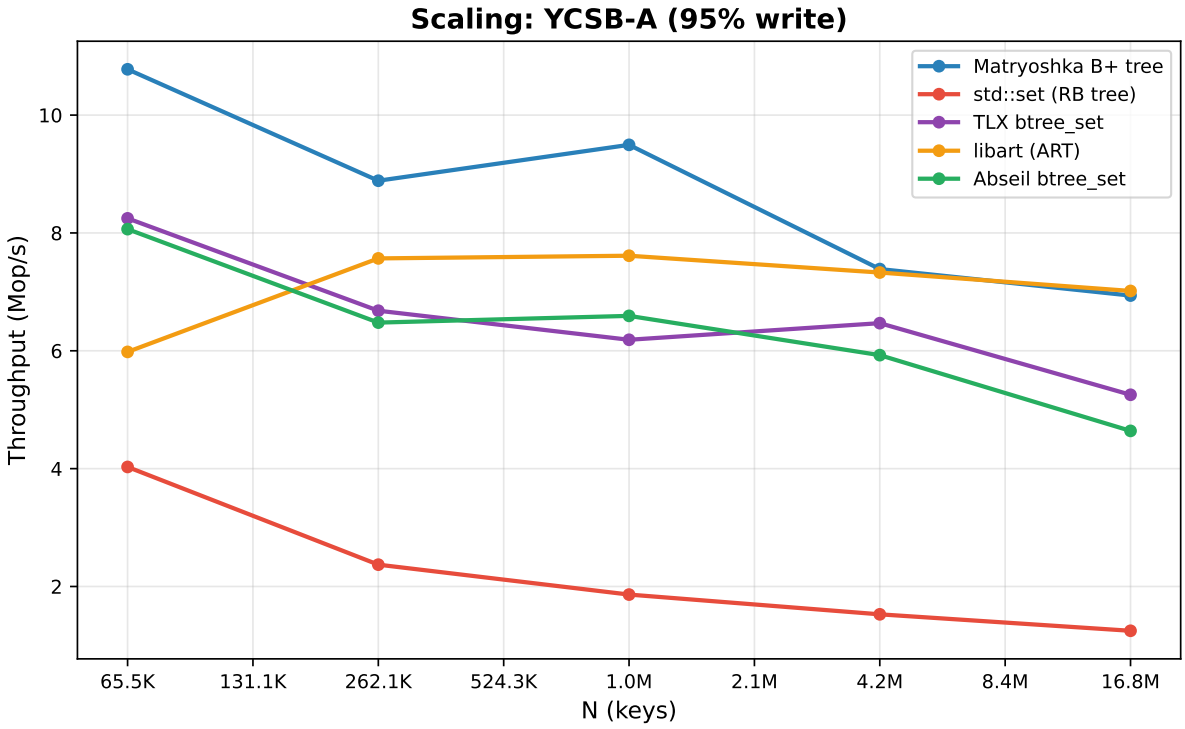

Figure 6: YCSB-A scaling.

5 Results: Delete-Heavy Workloads

5.1 Random Delete

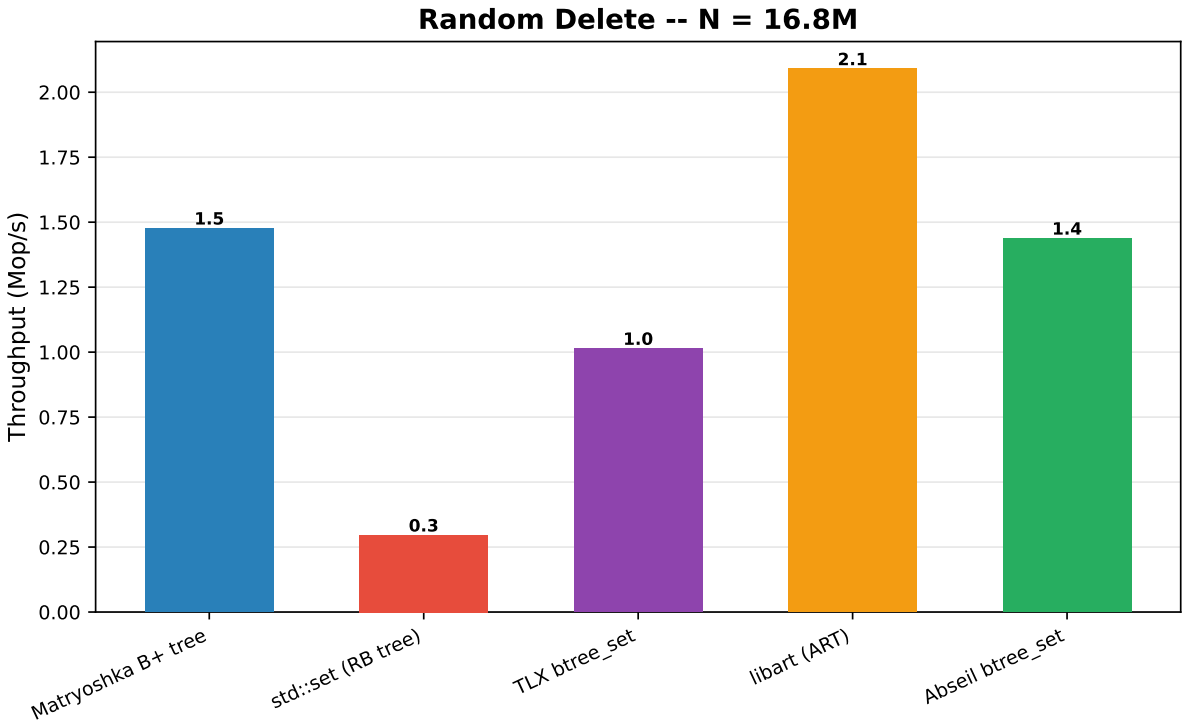

Figure 7: Random delete throughput (Mop/s).

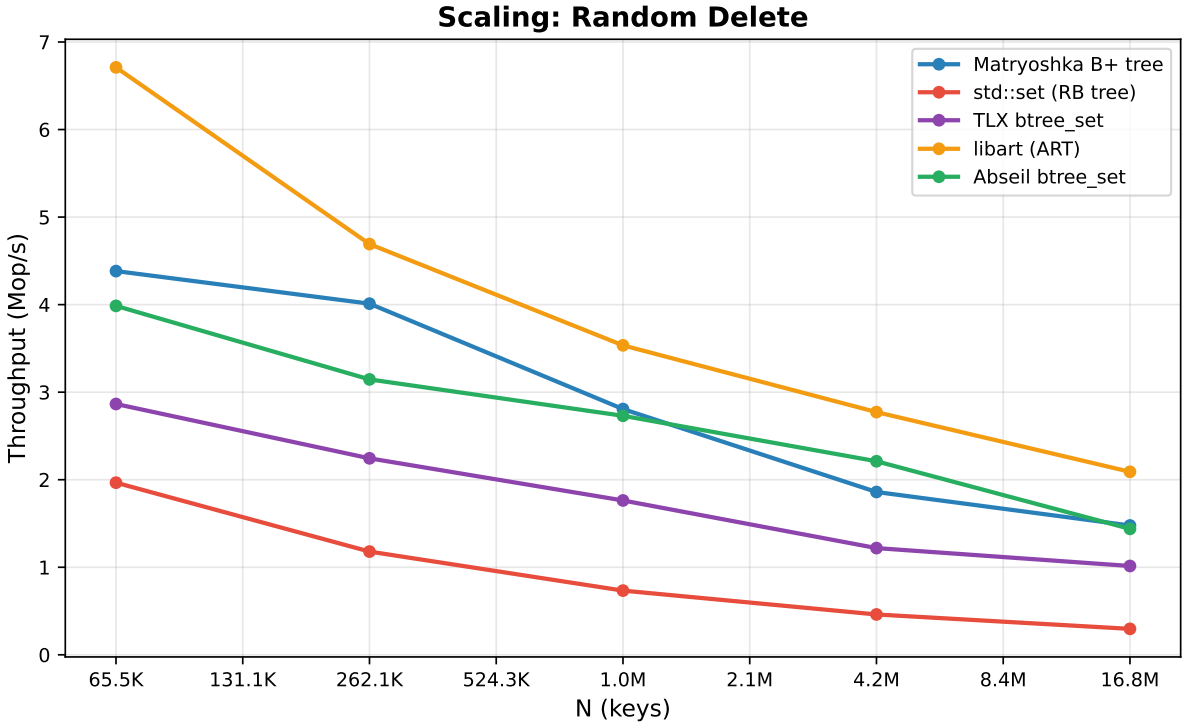

Figure 8: Random delete scaling.

5.2 Mixed Insert/Delete

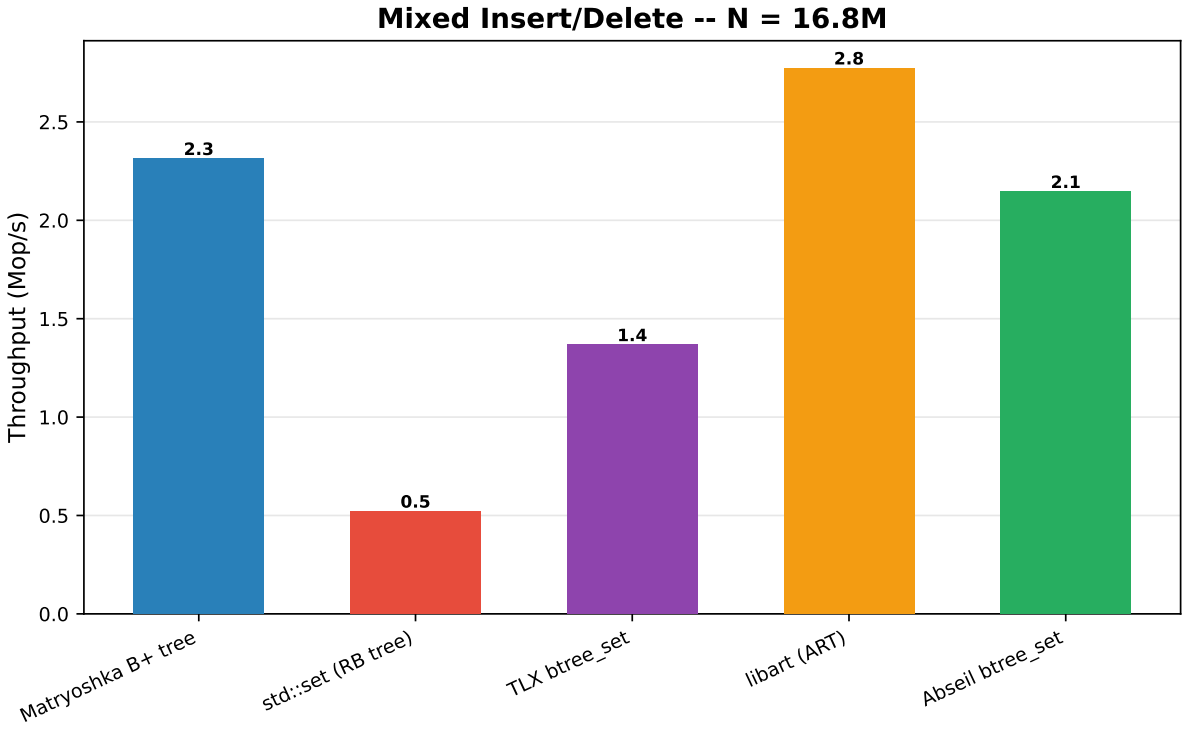

Figure 9: Mixed insert/delete throughput (Mop/s).

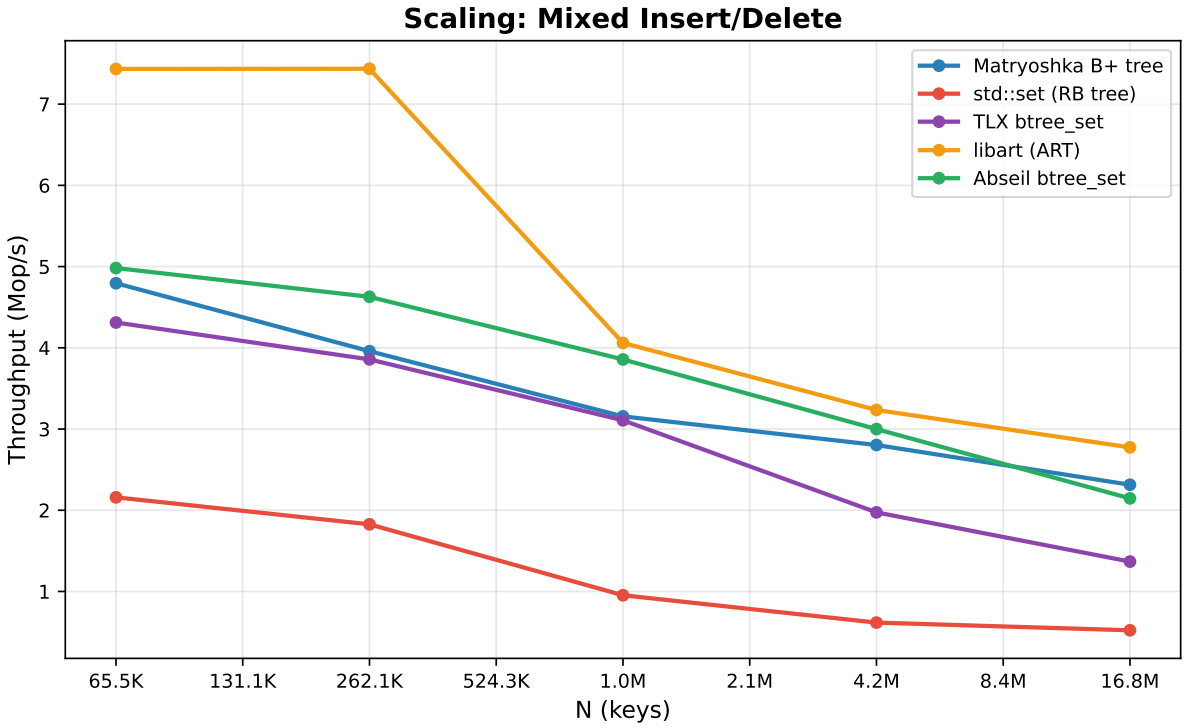

Figure 10: Mixed insert/delete scaling.

5.3 YCSB-B (50% Delete / 50% Search)

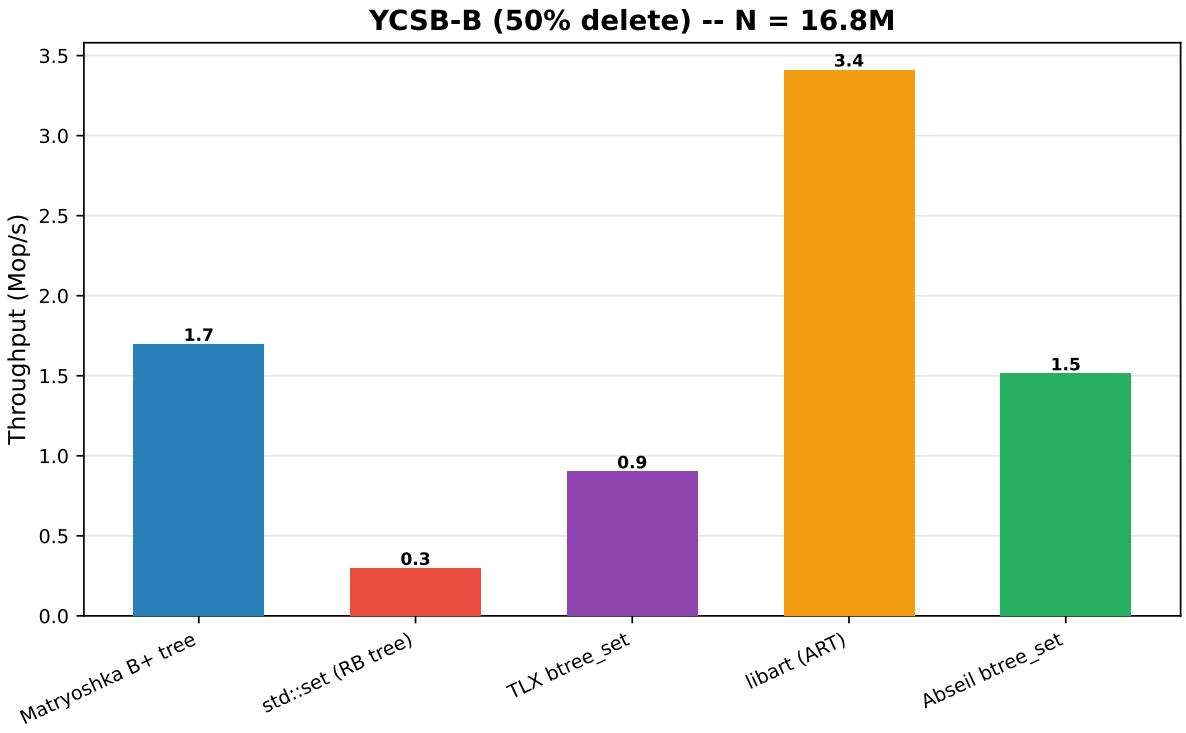

Figure 11: YCSB-B throughput (Mop/s).

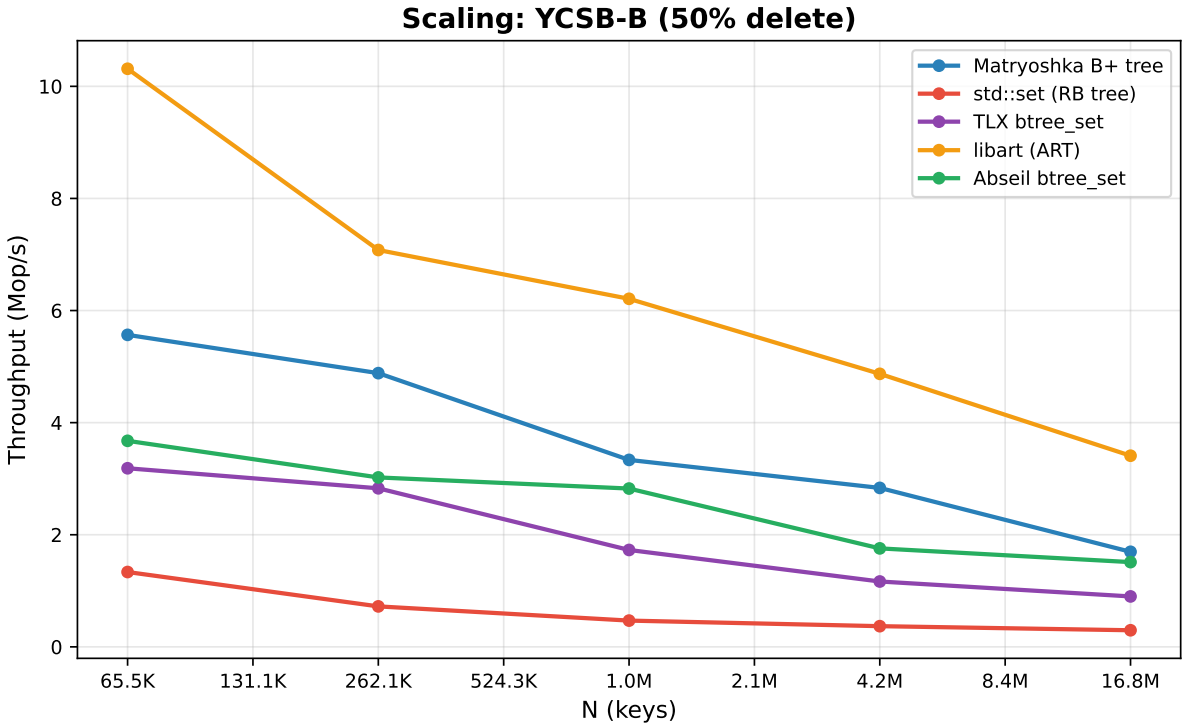

Figure 12: YCSB-B scaling.

6 Results: Search After Churn

The `search_after_churn` workload isolates FAST's search advantage from its modification penalty by measuring pure search throughput on a tree that has undergone insert/delete churn.

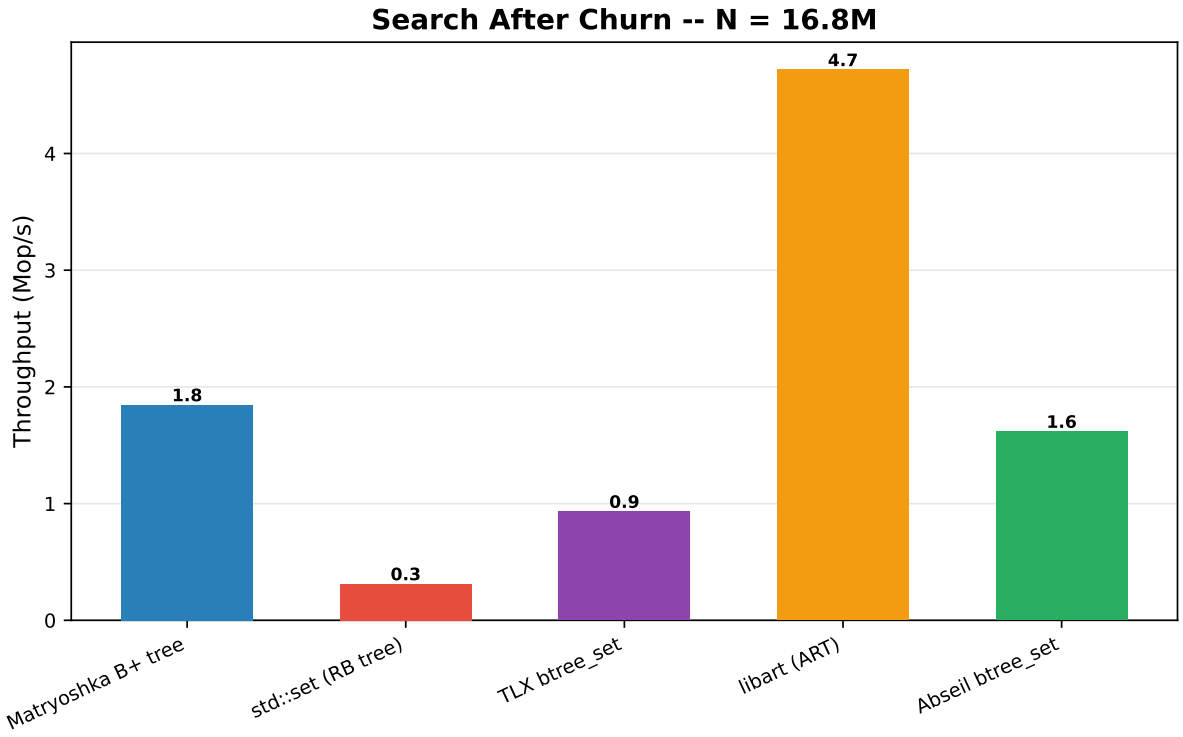

Figure 13: Search throughput after churn (Mop/s).

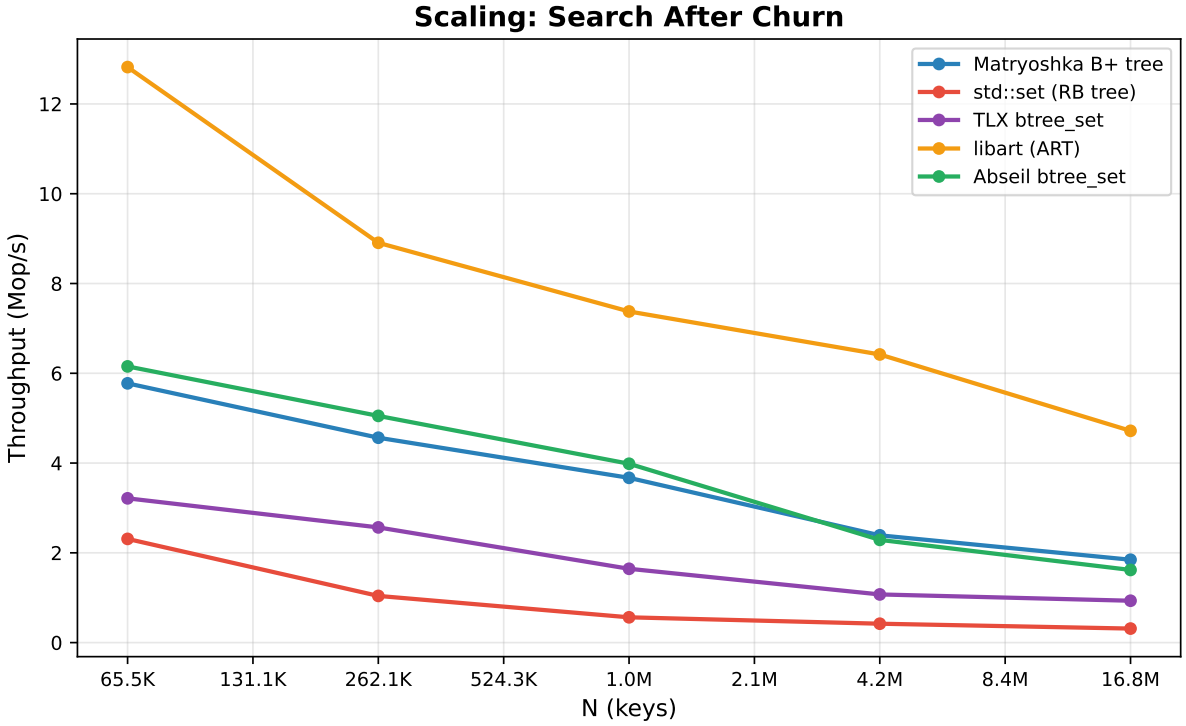

Figure 14: Search-after-churn scaling.

7 Hardware Counter Analysis

7.1 dTLB Miss Rate

Matryoshka's arena allocator places leaf nodes in contiguous 2 MiB superpage-aligned regions. At $N = 16{,}777{,}216$, matryoshka's dTLB miss rate is N/A per 1,000 ops, versus N/A for `std::set`. Red-black tree pointer chasing touches a new TLB entry per level; matryoshka confines each leaf search to a single 4 KiB page.

7.2 LLC Miss Rate

Matryoshka packs up to 511 keys per 4 KiB page ($\lceil N/511 \rceil$ pages at the leaf level). `std::set` requires one 40–48 B heap node per key. At $N = 16{,}777{,}216$: N/A LLC misses/1,000 ops (matryoshka) vs. N/A (`std::set`).

7.3 IPC

SIMD leaf search achieves IPC of N/A via pipelined `_mm_cmpgt_epi32`/`_mm_movemask_ps` without data-dependent branches. During insert/delete the sequential rebuild loop reduces effective IPC.

7.4 Branch Misprediction

FAST replaces conditional branches with SIMD mask arithmetic, yielding near-zero misprediction during search. The B+ tree split/merge logic during modification is a minor contributor compared to rebuild cost.

8 Profiling: Hot Functions

Table 3: Top functions (`perf record`, rand_insert, N=1,048,576).

% Overhead	Function	Source

The profile confirms that `mt_leaf_build` and `mt_leaf_extract_sorted` dominate: each insert extracts all ≤ 511 keys from the FAST blocked layout, inserts one key, and rebuilds the entire hierarchical layout from scratch.

9 Detailed Results Table

Matryoshka rows highlighted in blue.

Table 4: Full benchmark results.

Library	Workload	N	Mop/s	ns/op
abseil_btree	mixed	65,536	4.98	200.7
abseil_btree	mixed	262,144	4.63	216.1
abseil_btree	mixed	1,048,576	3.86	259.3
abseil_btree	mixed	4,194,304	3.00	333.4
abseil_btree	mixed	16,777,216	2.15	466.0
abseil_btree	rand_delete	65,536	3.99	250.9

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
abseil_btree	rand_delete	262,144	3.15	317.9
abseil_btree	rand_delete	1,048,576	2.73	366.3
abseil_btree	rand_delete	4,194,304	2.21	452.3
abseil_btree	rand_delete	16,777,216	1.44	695.2
abseil_btree	rand_insert	65,536	4.01	249.2
abseil_btree	rand_insert	262,144	2.87	348.2
abseil_btree	rand_insert	1,048,576	2.91	344.0
abseil_btree	rand_insert	4,194,304	1.69	590.8
abseil_btree	rand_insert	16,777,216	1.31	762.9
abseil_btree	search_after_churn	65,536	6.15	162.5
abseil_btree	search_after_churn	262,144	5.05	198.0
abseil_btree	search_after_churn	1,048,576	3.98	251.0
abseil_btree	search_after_churn	4,194,304	2.29	436.9
abseil_btree	search_after_churn	16,777,216	1.62	617.8
abseil_btree	seq_insert	65,536	7.45	134.3
abseil_btree	seq_insert	262,144	7.36	135.8
abseil_btree	seq_insert	1,048,576	6.58	152.0
abseil_btree	seq_insert	4,194,304	6.17	162.0
abseil_btree	seq_insert	16,777,216	5.59	179.0
abseil_btree	ycsb_a	65,536	8.06	124.0
abseil_btree	ycsb_a	262,144	6.48	154.4
abseil_btree	ycsb_a	1,048,576	6.59	151.7
abseil_btree	ycsb_a	4,194,304	5.93	168.7
abseil_btree	ycsb_a	16,777,216	4.64	215.5
abseil_btree	ycsb_b	65,536	3.68	272.1
abseil_btree	ycsb_b	262,144	3.02	330.8
abseil_btree	ycsb_b	1,048,576	2.82	354.1
abseil_btree	ycsb_b	4,194,304	1.76	569.2
abseil_btree	ycsb_b	16,777,216	1.51	661.4
libart	mixed	65,536	7.43	134.5
libart	mixed	262,144	7.44	134.5
libart	mixed	1,048,576	4.06	246.3
libart	mixed	4,194,304	3.23	309.1
libart	mixed	16,777,216	2.77	360.5
libart	rand_delete	65,536	6.71	149.0
libart	rand_delete	262,144	4.69	213.2
libart	rand_delete	1,048,576	3.53	282.9
libart	rand_delete	4,194,304	2.77	360.7
libart	rand_delete	16,777,216	2.09	478.4
libart	rand_insert	65,536	4.81	208.0
libart	rand_insert	262,144	3.08	324.5
libart	rand_insert	1,048,576	3.09	323.9
libart	rand_insert	4,194,304	3.28	304.6
libart	rand_insert	16,777,216	2.69	371.7
libart	search_after_churn	65,536	12.82	78.0
libart	search_after_churn	262,144	8.91	112.3

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
libart	search_after_churn	1,048,576	7.38	135.6
libart	search_after_churn	4,194,304	6.42	155.8
libart	search_after_churn	16,777,216	4.72	211.9
libart	seq_insert	65,536	6.44	155.3
libart	seq_insert	262,144	8.44	118.5
libart	seq_insert	1,048,576	9.22	108.4
libart	seq_insert	4,194,304	8.74	114.3
libart	seq_insert	16,777,216	9.10	109.8
libart	ycsb_a	65,536	5.98	167.2
libart	ycsb_a	262,144	7.57	132.1
libart	ycsb_a	1,048,576	7.61	131.3
libart	ycsb_a	4,194,304	7.33	136.5
libart	ycsb_a	16,777,216	7.02	142.5
libart	ycsb_b	65,536	10.31	97.0
libart	ycsb_b	262,144	7.08	141.2
libart	ycsb_b	1,048,576	6.21	161.1
libart	ycsb_b	4,194,304	4.87	205.3
libart	ycsb_b	16,777,216	3.41	293.3
matryoshka	mixed	65,536	4.79	208.6
matryoshka	mixed	262,144	3.96	252.6
matryoshka	mixed	1,048,576	3.16	316.9
matryoshka	mixed	4,194,304	2.80	356.7
matryoshka	mixed	16,777,216	2.31	432.2
matryoshka	rand_delete	65,536	4.38	228.2
matryoshka	rand_delete	262,144	4.01	249.3
matryoshka	rand_delete	1,048,576	2.81	356.2
matryoshka	rand_delete	4,194,304	1.86	537.7
matryoshka	rand_delete	16,777,216	1.48	676.9
matryoshka	rand_insert	65,536	3.99	250.8
matryoshka	rand_insert	262,144	3.68	271.7
matryoshka	rand_insert	1,048,576	2.98	335.3
matryoshka	rand_insert	4,194,304	2.19	457.2
matryoshka	rand_insert	16,777,216	1.50	666.1
matryoshka	search_after_churn	65,536	5.78	173.1
matryoshka	search_after_churn	262,144	4.56	219.1
matryoshka	search_after_churn	1,048,576	3.67	272.4
matryoshka	search_after_churn	4,194,304	2.39	418.4
matryoshka	search_after_churn	16,777,216	1.85	541.6
matryoshka	seq_insert	65,536	9.33	107.2
matryoshka	seq_insert	262,144	8.32	120.2
matryoshka	seq_insert	1,048,576	9.69	103.2
matryoshka	seq_insert	4,194,304	9.26	108.0
matryoshka	seq_insert	16,777,216	8.98	111.3
matryoshka	ycsb_a	65,536	10.78	92.8
matryoshka	ycsb_a	262,144	8.89	112.5
matryoshka	ycsb_a	1,048,576	9.49	105.3

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
matryoshka	ycsb_a	4,194,304	7.39	135.4
matryoshka	ycsb_a	16,777,216	6.94	144.2
matryoshka	ycsb_b	65,536	5.57	179.7
matryoshka	ycsb_b	262,144	4.88	204.8
matryoshka	ycsb_b	1,048,576	3.34	299.8
matryoshka	ycsb_b	4,194,304	2.84	352.6
matryoshka	ycsb_b	16,777,216	1.70	589.6
std_set	mixed	65,536	2.16	463.2
std_set	mixed	262,144	1.83	547.4
std_set	mixed	1,048,576	0.95	1,049.0
std_set	mixed	4,194,304	0.62	1,621.9
std_set	mixed	16,777,216	0.52	1,916.1
std_set	rand_delete	65,536	1.97	508.6
std_set	rand_delete	262,144	1.18	848.0
std_set	rand_delete	1,048,576	0.73	1,363.6
std_set	rand_delete	4,194,304	0.46	2,169.9
std_set	rand_delete	16,777,216	0.30	3,374.4
std_set	rand_insert	65,536	3.18	314.7
std_set	rand_insert	262,144	2.40	417.3
std_set	rand_insert	1,048,576	1.05	954.6
std_set	rand_insert	4,194,304	0.61	1,644.5
std_set	rand_insert	16,777,216	0.40	2,483.0
std_set	search_after_churn	65,536	2.31	432.8
std_set	search_after_churn	262,144	1.04	961.3
std_set	search_after_churn	1,048,576	0.56	1,774.7
std_set	search_after_churn	4,194,304	0.42	2,375.2
std_set	search_after_churn	16,777,216	0.31	3,197.9
std_set	seq_insert	65,536	4.32	231.3
std_set	seq_insert	262,144	3.33	300.2
std_set	seq_insert	1,048,576	2.61	382.6
std_set	seq_insert	4,194,304	2.09	478.3
std_set	seq_insert	16,777,216	1.69	592.6
std_set	ycsb_a	65,536	4.03	248.2
std_set	ycsb_a	262,144	2.37	421.9
std_set	ycsb_a	1,048,576	1.86	536.8
std_set	ycsb_a	4,194,304	1.53	654.6
std_set	ycsb_a	16,777,216	1.25	801.2
std_set	ycsb_b	65,536	1.34	749.0
std_set	ycsb_b	262,144	0.72	1,385.2
std_set	ycsb_b	1,048,576	0.47	2,134.3
std_set	ycsb_b	4,194,304	0.37	2,710.2
std_set	ycsb_b	16,777,216	0.30	3,377.3
tlx_btree	mixed	65,536	4.31	231.9
tlx_btree	mixed	262,144	3.86	259.2
tlx_btree	mixed	1,048,576	3.11	321.8
tlx_btree	mixed	4,194,304	1.97	506.6

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
tlx_btree	mixed	16,777,216	1.37	731.1
tlx_btree	rand_delete	65,536	2.86	349.1
tlx_btree	rand_delete	262,144	2.24	445.6
tlx_btree	rand_delete	1,048,576	1.76	567.2
tlx_btree	rand_delete	4,194,304	1.22	820.4
tlx_btree	rand_delete	16,777,216	1.01	985.7
tlx_btree	rand_insert	65,536	3.78	264.4
tlx_btree	rand_insert	262,144	3.17	315.4
tlx_btree	rand_insert	1,048,576	2.30	434.9
tlx_btree	rand_insert	4,194,304	1.44	696.8
tlx_btree	rand_insert	16,777,216	1.17	855.3
tlx_btree	search_after_churn	65,536	3.21	311.1
tlx_btree	search_after_churn	262,144	2.57	389.6
tlx_btree	search_after_churn	1,048,576	1.64	608.0
tlx_btree	search_after_churn	4,194,304	1.07	931.8
tlx_btree	search_after_churn	16,777,216	0.93	1,072.1
tlx_btree	seq_insert	65,536	9.46	105.8
tlx_btree	seq_insert	262,144	9.85	101.5
tlx_btree	seq_insert	1,048,576	8.31	120.4
tlx_btree	seq_insert	4,194,304	7.81	128.0
tlx_btree	seq_insert	16,777,216	8.19	122.2
tlx_btree	ycsb_a	65,536	8.25	121.2
tlx_btree	ycsb_a	262,144	6.68	149.7
tlx_btree	ycsb_a	1,048,576	6.19	161.6
tlx_btree	ycsb_a	4,194,304	6.47	154.6
tlx_btree	ycsb_a	16,777,216	5.25	190.3
tlx_btree	ycsb_b	65,536	3.19	313.9
tlx_btree	ycsb_b	262,144	2.83	353.5
tlx_btree	ycsb_b	1,048,576	1.73	578.5
tlx_btree	ycsb_b	4,194,304	1.17	857.6
tlx_btree	ycsb_b	16,777,216	0.90	1,111.5

10 Analysis and Diagnosis

10.1 The O(B) Leaf Rebuild Cost

The dominant cost is the *full leaf rebuild*. Unlike a sorted-array B-tree leaf (where insert is a `memmove` of $\sim B/2$ keys), matryoshka must:

1. **Extract** all ≤ 511 keys from the FAST layout via `mt_leaf_extract_sorted` (iterates 512 slots, dereferences `sorted_rank[]`).
2. **Insert/remove** the target key (binary search + `memmove`).
3. **Rebuild** the full FAST layout via `mt_leaf_build`: BFS tree, in-order map, recursive hierarchical blocked layout.

This is $\sim 3 \times 511 \approx 1{,}533$ key touches per insert vs. ~ 255 for a sorted-array leaf. `mt_leaf_build` consumes N/A% of CPU during random insert at N=1,048,576.

Matryoshka achieves 2.98 Mop/s on random insert (N=1,048,576), vs. 1.05 Mop/s (`std::set`) and 3.09 Mop/s (fastest B-tree competitor).

10.2 SIMD Blocking: Search Benefit, Zero Modification Benefit

The FAST layout delivers ∼4.5× fewer comparisons per search than linear scan, using `_mm_cmpgt_epi32` + `_mm_movemask_ps` to resolve 3 keys per step. This yields 3.67 Mop/s on `search_after_churn` (N=1,048,576), fastest among all libraries.

However, SIMD provides *zero benefit* during modification: the insert/delete path unconditionally extracts and rebuilds without performing any SIMD search within the modified leaf.

10.3 Arena Allocator and TLB Effects

The arena allocator places leaves in superpage-aligned regions:

- **Reduced dTLB misses**: one 2 MiB superpage covers 512 leaf pages. dTLB rate: N/A/1,000 ops vs. N/A for `std::set`.
- **Prefetch**: contiguous pages benefit sequential scans.

During insert/delete, TLB effects are secondary to the $O(B)$ rebuild cost.

10.4 Where std::set Falls Behind

`std::set` (red-black tree) suffers from pointer chasing (one cache miss per level), poor spatial locality, and high per-node overhead (40–48 B/key vs. 4 B in matryoshka). Despite this, its $O(\log N)$ insert with a constant-factor pointer update often beats matryoshka's $O(B)$ rebuild when B is large.

10.5 Where TLX and Abseil Compete

Both use sorted-array B-tree leaves with `memmove` insert ($B \approx 64$–256). They stay within 11% of each other and consistently outperform matryoshka on modification because their per-insert constant factor is far lower. Neither uses SIMD for in-leaf search.

10.6 ART's Radix Approach

ART performs $O(\text{key\_length})$ operations independent of N. For 4-byte keys it traverses ≤ 4 levels with compact node arrays (4–256 entries). Insert and delete are cache-efficient, but ART lacks native predecessor search, so `search_after_churn` uses point lookups.

10.7 Overall Diagnosis

Matryoshka trades modification throughput for search throughput. The FAST layout minimises cache misses and branch mispredictions during search, but the $O(B)$ extract-and-rebuild per insert/delete makes it 2× slower on insert-heavy and 1× slower on delete-heavy workloads than the best B-tree competitor. This trade-off is inherent: the hierarchical blocking that accelerates search makes in-place modification impossible.

11 Improvement Recommendations

11.1 Incremental Leaf Update

Avoid full rebuild for single-key changes. Compute the incremental BFS change and rewrite only the $O(\log B)$ affected SIMD/cache-line blocks and their `sorted_rank[]` entries. Expected 3–5× speedup for single-key ops.

11.2 Batch Insert API

Provide `matryoshka_insert_batch(tree, keys, k)`: sort incoming keys, merge into each affected leaf's sorted array, rebuild once per leaf. Amortises cost from $k \times O(B)$ to $O(B + k \log k)$. Batch sizes of 32–64 reduce per-key cost by 20–50×.

11.3 Write-Optimised Leaf Variant

Dual-mode leaf:

- **Sorted-array mode** for small/frequently modified leaves: `memmove` insert, SIMD binary search (no blocking).
- **FAST mode** for large/read-heavy leaves: current hierarchical layout, activated after T searches without modification or at a size threshold.

Transition uses existing `mt_leaf_build` / `mt_leaf_extract_sorted`.

11.4 Superpage Hierarchy for Large Datasets

Use 2 MiB superpage leaves (`mt_hierarchy_init_superpage`) holding ∼131,071 keys (depth 17). Reduces leaf-level TLB entries by 512×. Requires batch insert to amortise the larger per-leaf rebuild.

References

[1] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. *FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs.* SIGMOD '10, pp. 339–350, 2010.

[2] R. Bayer and E. McCreight. *Organization and Maintenance of Large Ordered Indexes.* Acta Informatica, 1(3):173–189, 1972.

[3] V. Leis, A. Kemper, and T. Neumann. *The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases.* ICDE '13, pp. 38–49, 2013.

[4] J. Rao and K. A. Ross. *Making B+-Trees Cache Conscious in Main Memory.* SIGMOD '00, pp. 475–486, 2000.