Matryoshka B+ Tree: Insert/Delete Performance Report

Comparative Benchmark Results

2026-02-20T08:33:24

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 B+ tree whose page-sized leaf nodes contain nested B+ sub-trees of cache-line-sized (64 B) sub-nodes, with SIMD-accelerated search at every level — 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	B+ tree with nested CL sub-tree leaves (up to 855 keys/page), SIMD
`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≈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≈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

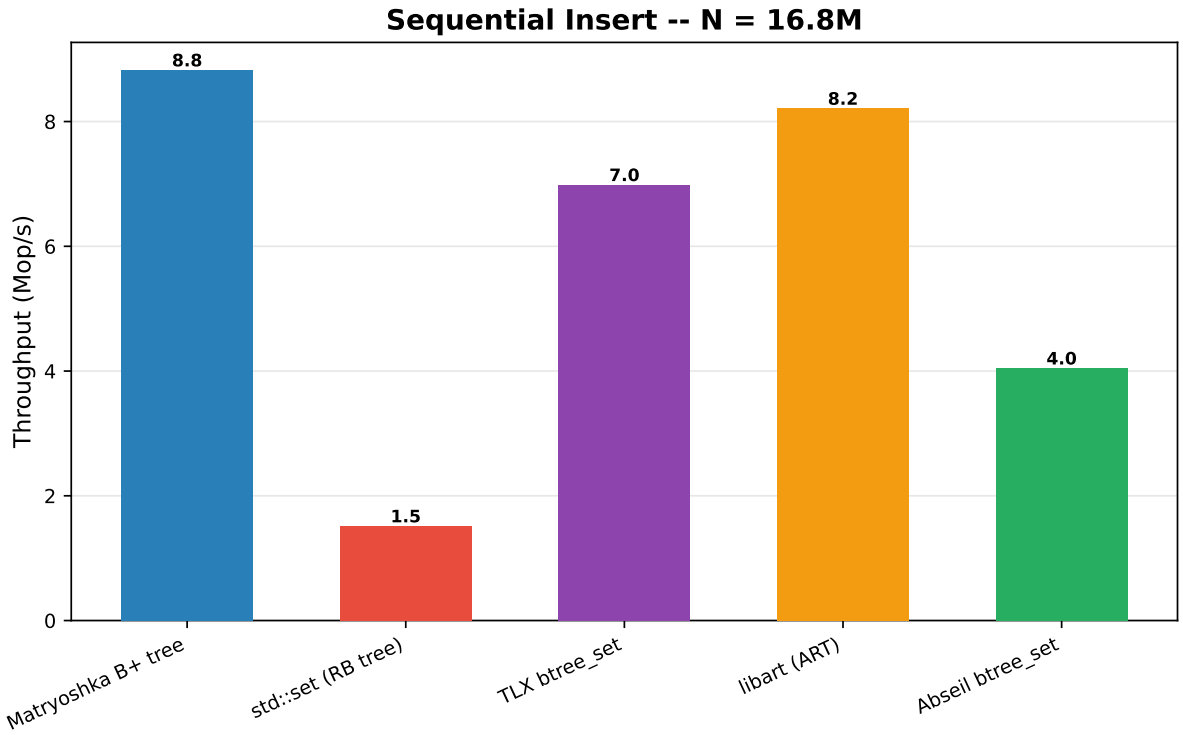

Figure 1: Sequential insert throughput (Mop/s).

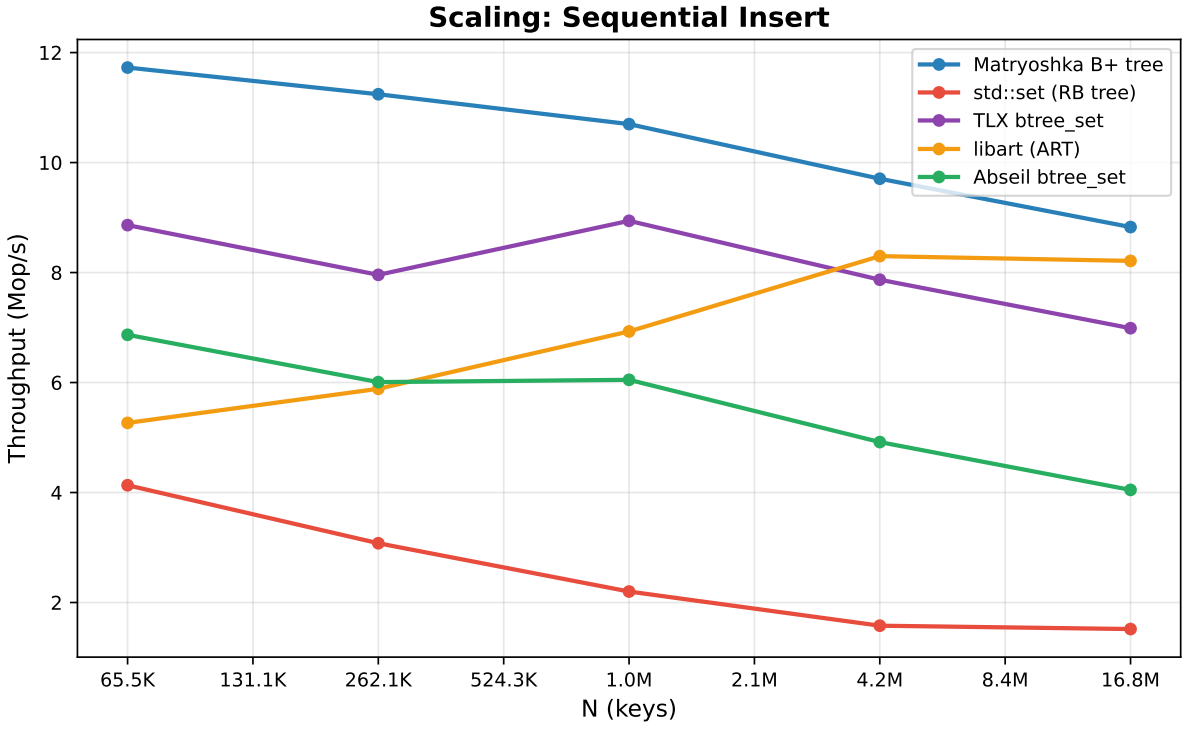

Figure 2: Sequential insert scaling.

4.2 Random Insert

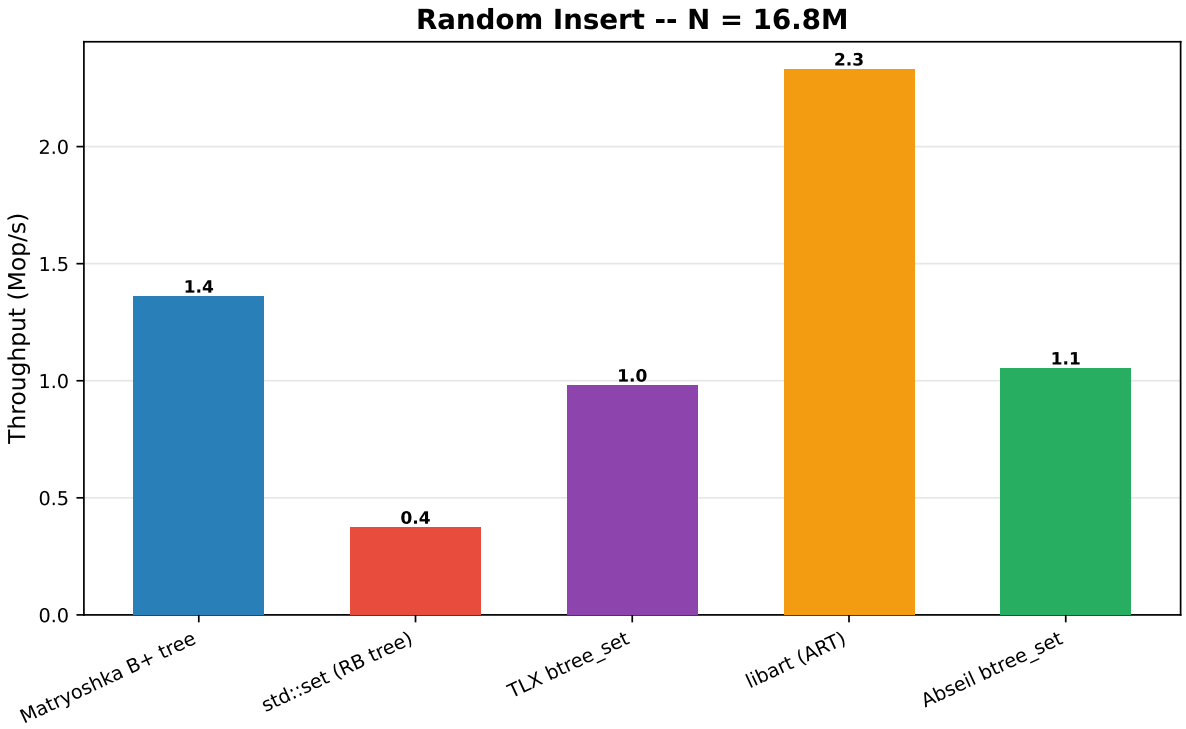

Figure 3: Random insert throughput (Mop/s).

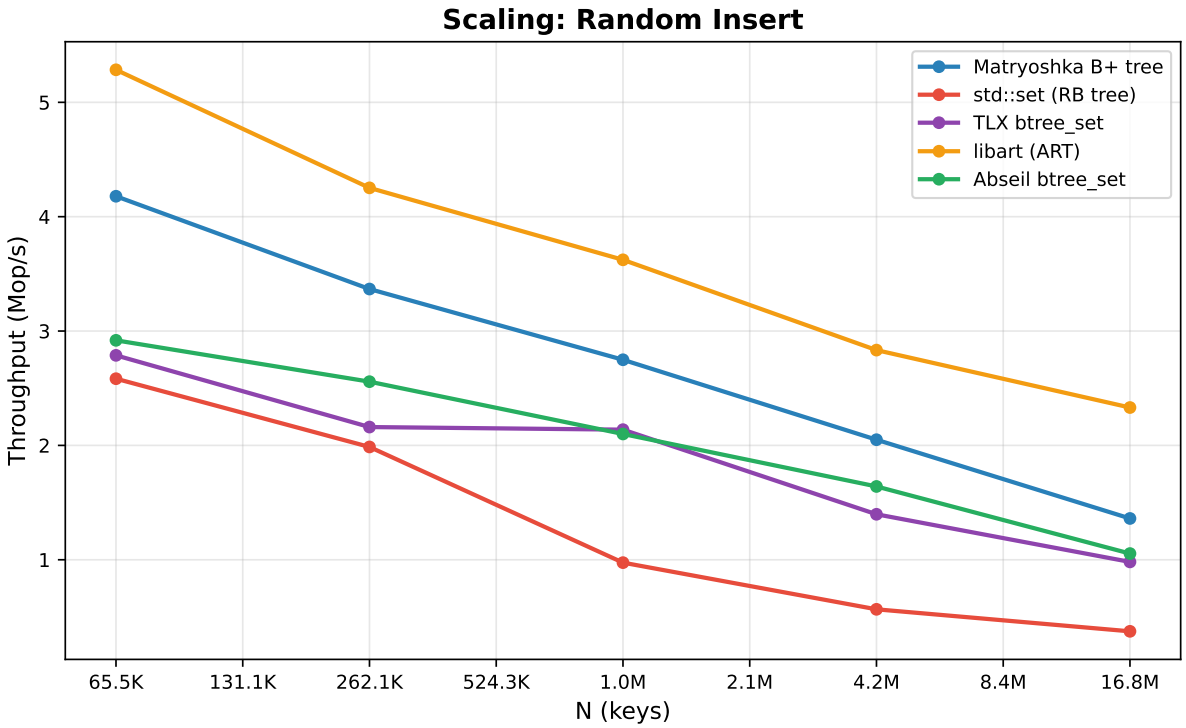

Figure 4: Random insert scaling.

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

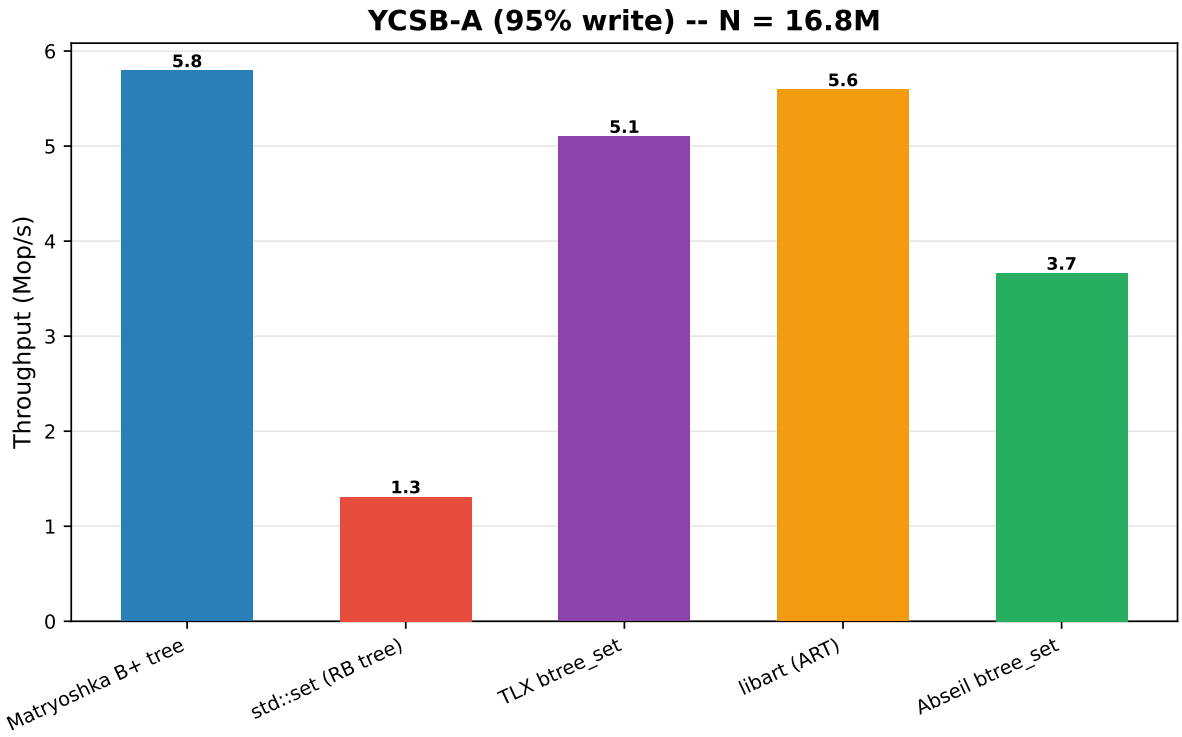

Figure 5: YCSB-A throughput (Mop/s).

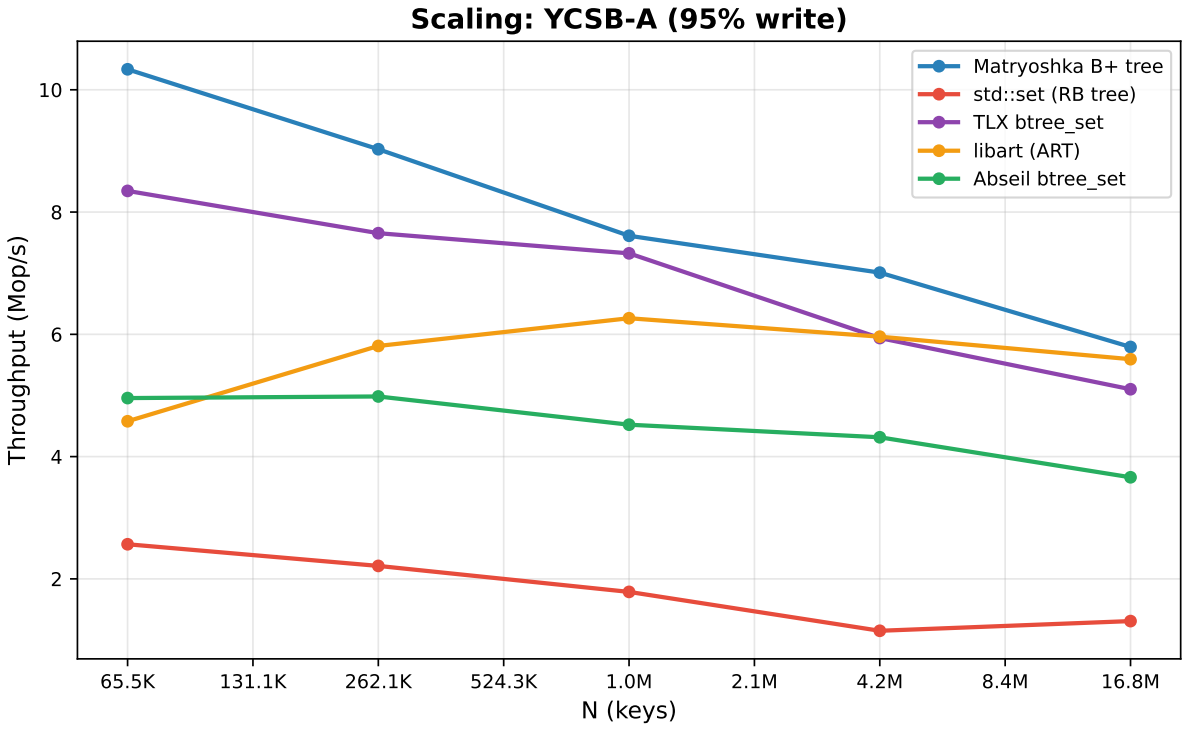

Figure 6: YCSB-A scaling.

5 Results: Delete-Heavy Workloads

5.1 Random Delete

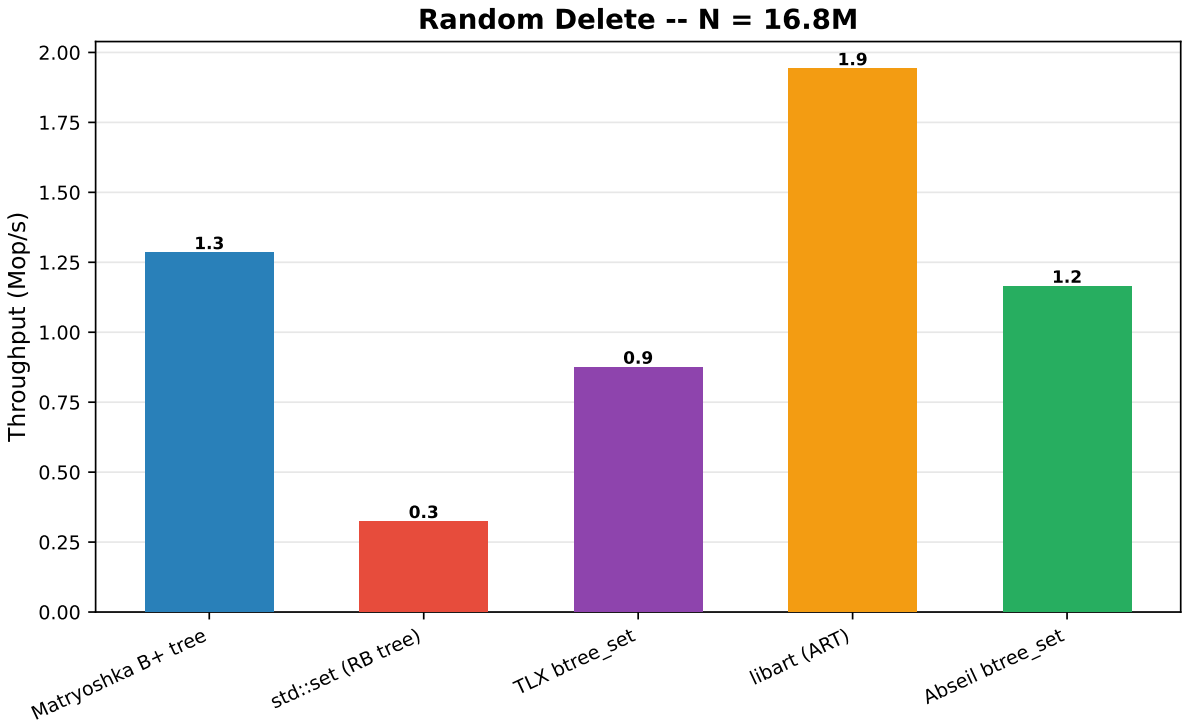

Figure 7: Random delete throughput (Mop/s).

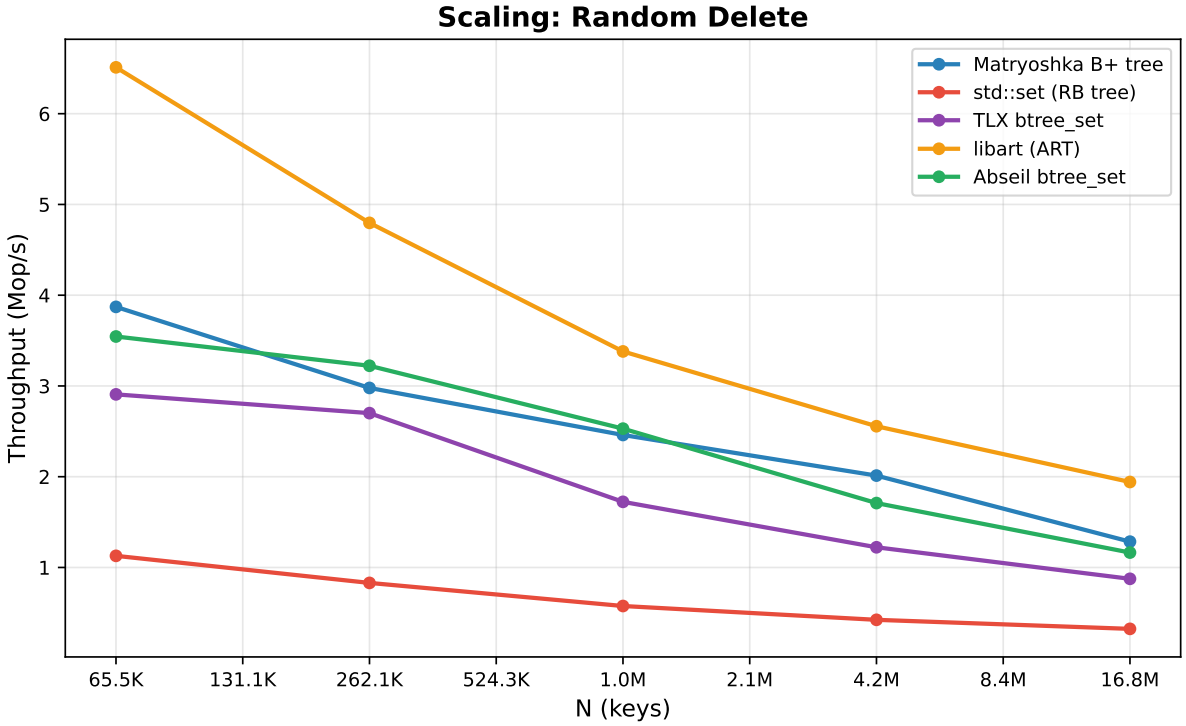

Figure 8: Random delete scaling.

5.2 Mixed Insert/Delete

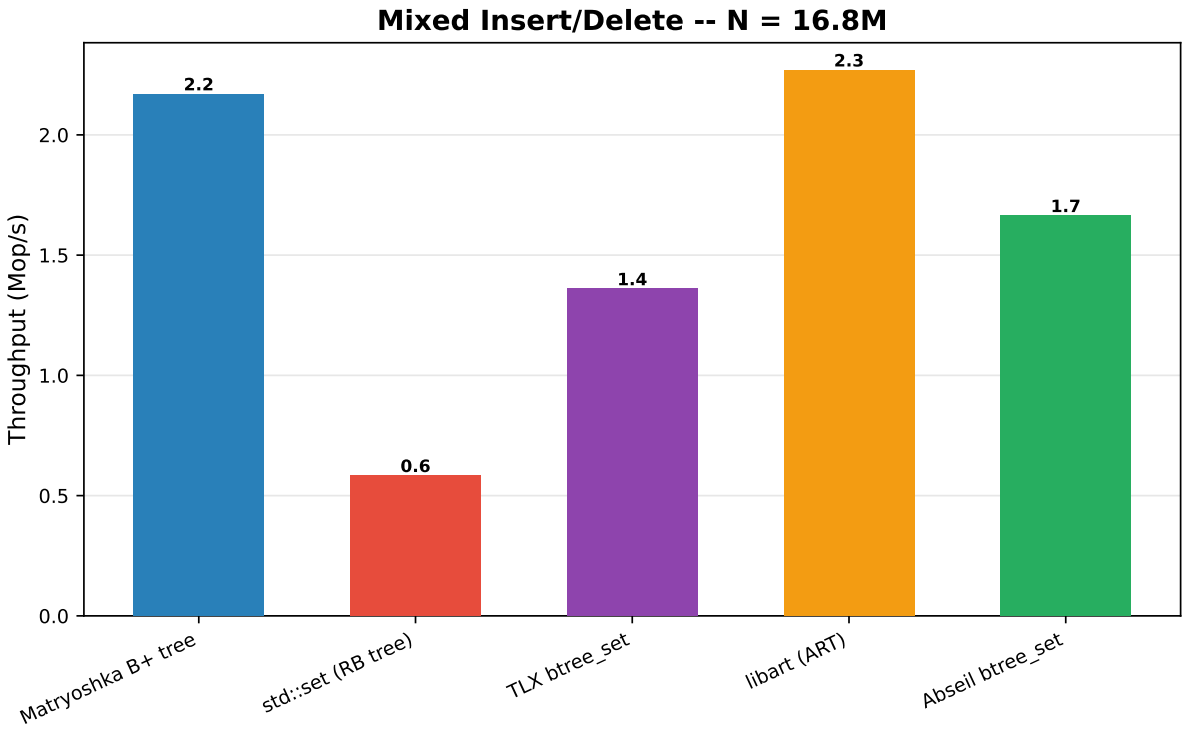

Figure 9: Mixed insert/delete throughput (Mop/s).

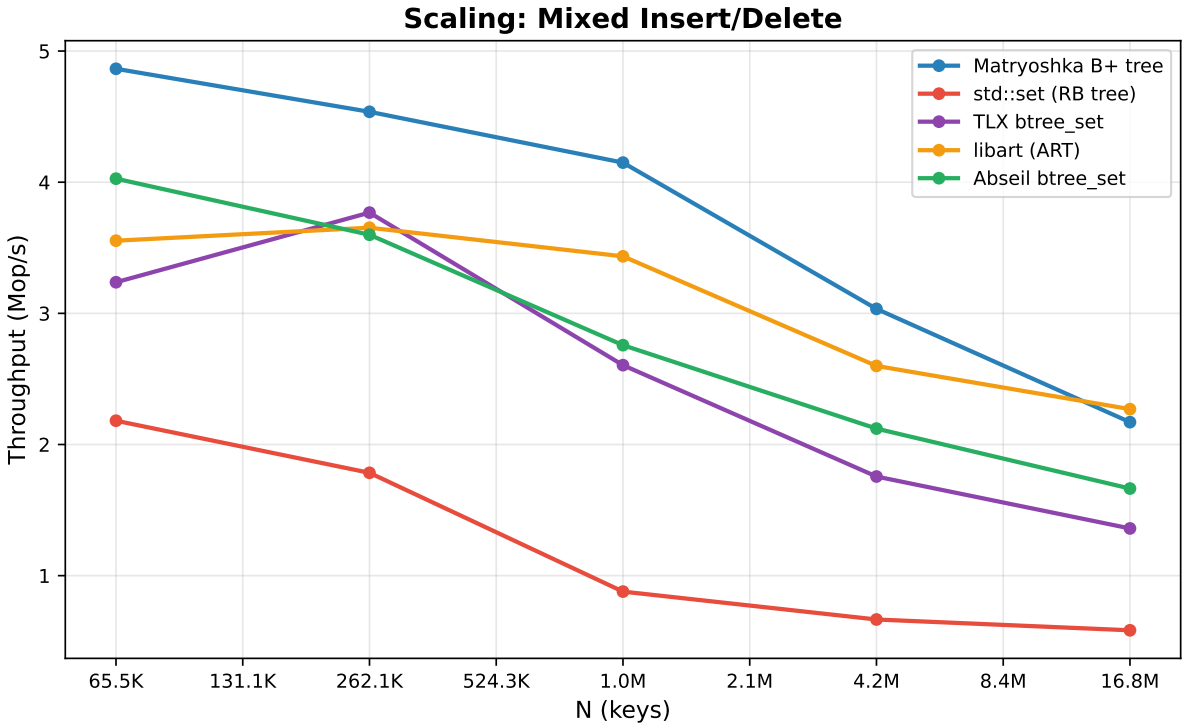

Figure 10: Mixed insert/delete scaling.

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

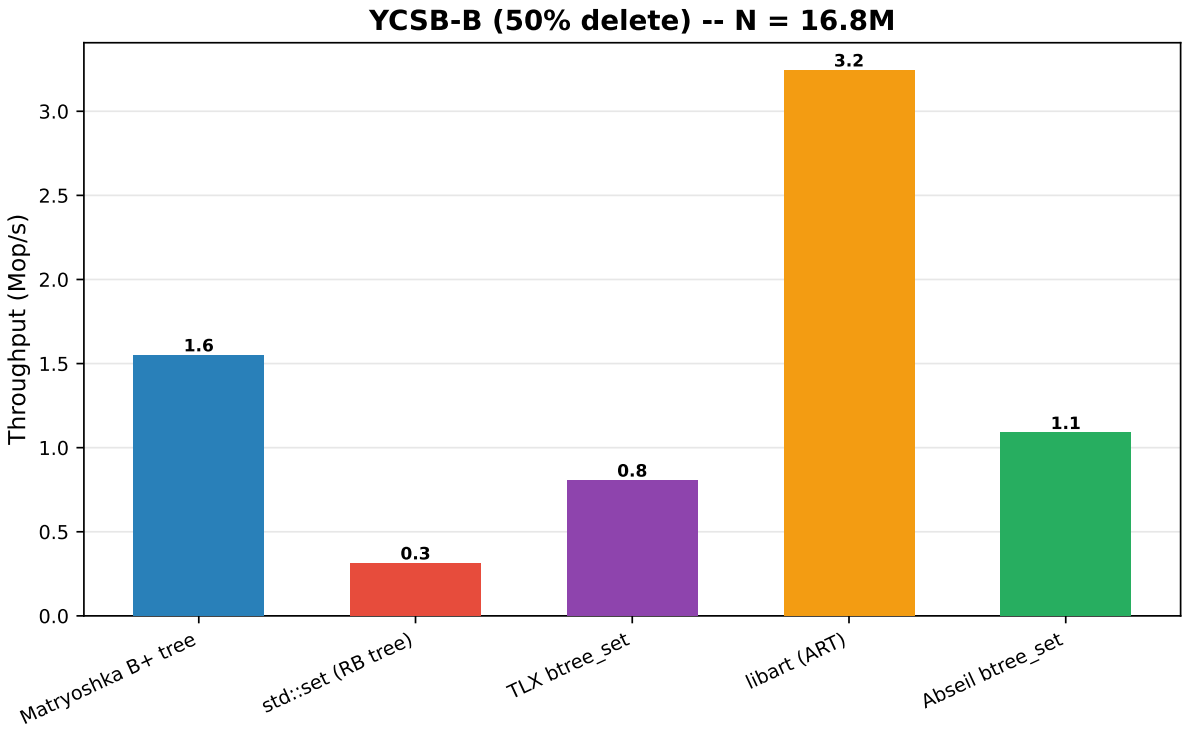

Figure 11: YCSB-B throughput (Mop/s).

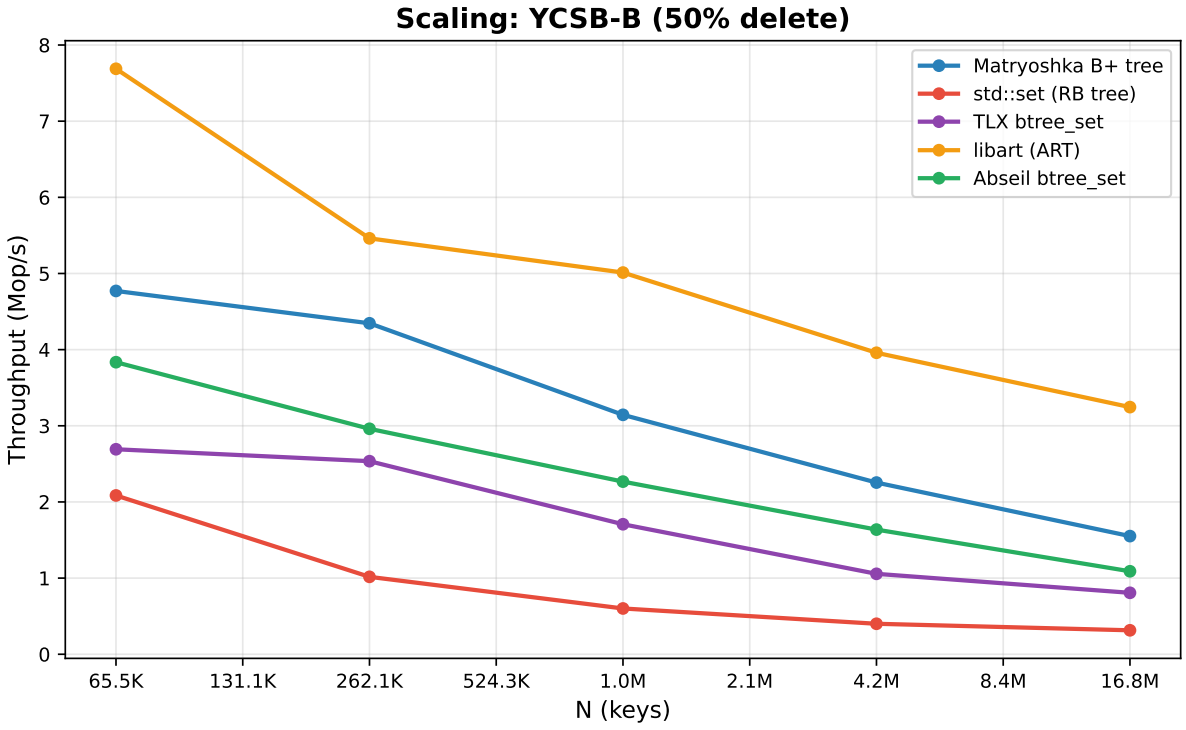

Figure 12: YCSB-B scaling.

6 Results: Search After Churn

The `search_after_churn` workload measures pure search throughput on a tree that has undergone insert/delete churn, isolating search performance from modification cost.

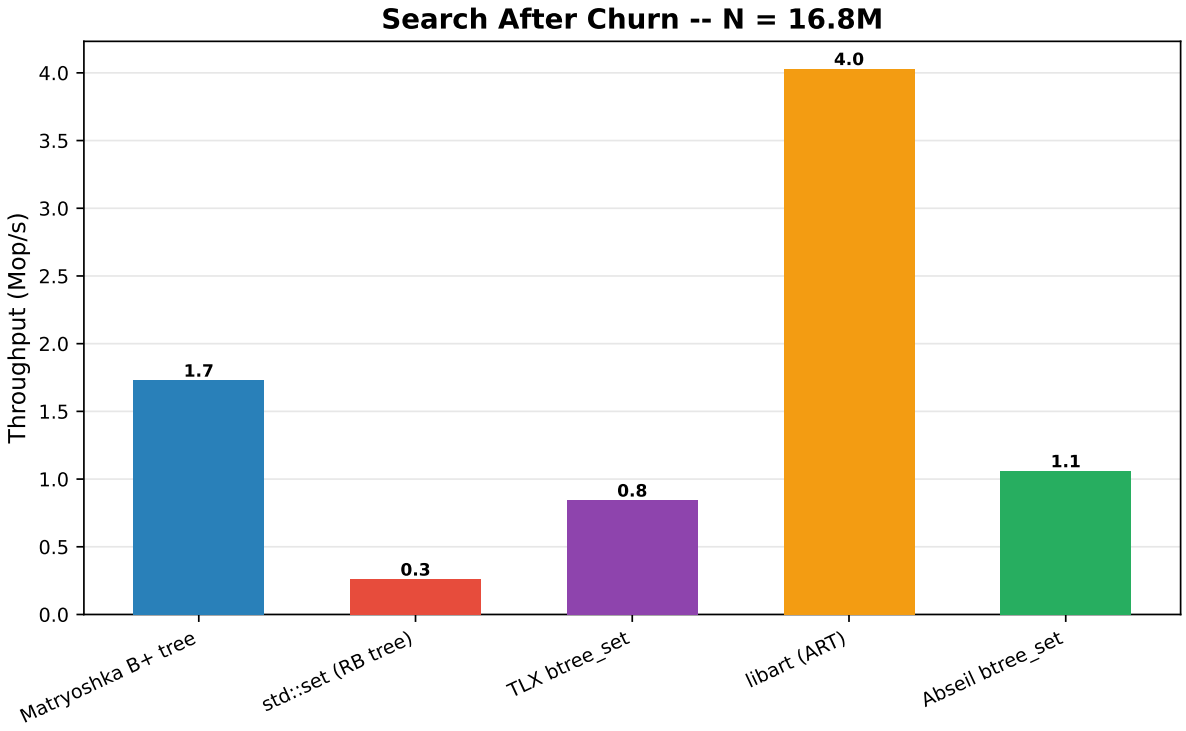

Figure 13: Search throughput after churn (Mop/s).

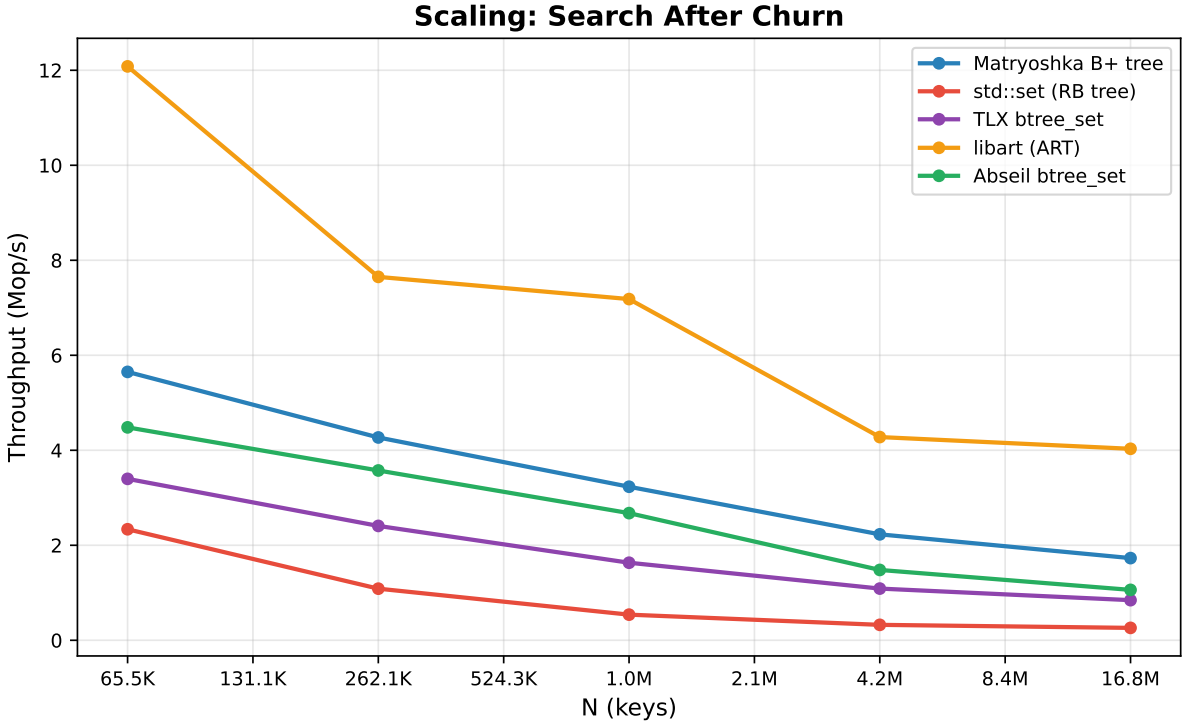

Figure 14: Search-after-churn scaling.

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 855 keys per 4 KiB page using a nested B+ sub-tree of cache-line sub-nodes. `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 search at every level (CL leaves, CL internals, outer internal nodes) achieves IPC of N/A via pipelined `_mm_cmpgt_epi32`/`_mm_movemask_ps` without data-dependent branches. Insert and delete paths also benefit from SIMD navigation through the CL sub-tree to locate the target sub-node.

7.4 Branch Misprediction

SIMD mask arithmetic replaces conditional branches during search, yielding near-zero misprediction. Modification operations navigate the CL sub-tree using the same SIMD search, with only the final CL leaf insert/delete using scalar `memmove` on ≤ 14 keys.

8 Profiling: Hot Functions

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

% Overhead	Function	Source

The hot functions are the page-level sub-tree operations: `mt_page_insert` and `mt_page_delete` navigate the CL sub-tree via SIMD search, then perform a scalar insert or delete within a single 64 B cache-line sub-node. CL sub-node splits and merges occur only on overflow or underflow.

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	4.03	248.3
abseil_btree	mixed	262,144	3.60	277.8
abseil_btree	mixed	1,048,576	2.76	362.7
abseil_btree	mixed	4,194,304	2.12	471.4

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
abseil_btree	mixed	16,777,216	1.66	600.9
abseil_btree	rand_delete	65,536	3.54	282.1
abseil_btree	rand_delete	262,144	3.22	310.3
abseil_btree	rand_delete	1,048,576	2.53	395.3
abseil_btree	rand_delete	4,194,304	1.71	585.0
abseil_btree	rand_delete	16,777,216	1.16	858.9
abseil_btree	rand_insert	65,536	2.92	342.6
abseil_btree	rand_insert	262,144	2.56	391.0
abseil_btree	rand_insert	1,048,576	2.10	476.2
abseil_btree	rand_insert	4,194,304	1.64	609.4
abseil_btree	rand_insert	16,777,216	1.05	948.6
abseil_btree	search_after_churn	65,536	4.48	223.1
abseil_btree	search_after_churn	262,144	3.57	279.8
abseil_btree	search_after_churn	1,048,576	2.68	373.6
abseil_btree	search_after_churn	4,194,304	1.48	675.3
abseil_btree	search_after_churn	16,777,216	1.06	942.5
abseil_btree	seq_insert	65,536	6.87	145.6
abseil_btree	seq_insert	262,144	6.01	166.4
abseil_btree	seq_insert	1,048,576	6.05	165.3
abseil_btree	seq_insert	4,194,304	4.92	203.4
abseil_btree	seq_insert	16,777,216	4.05	247.1
abseil_btree	ycsb_a	65,536	4.96	201.7
abseil_btree	ycsb_a	262,144	4.98	200.7
abseil_btree	ycsb_a	1,048,576	4.52	221.1
abseil_btree	ycsb_a	4,194,304	4.32	231.7
abseil_btree	ycsb_a	16,777,216	3.66	273.1
abseil_btree	ycsb_b	65,536	3.84	260.8
abseil_btree	ycsb_b	262,144	2.96	337.8
abseil_btree	ycsb_b	1,048,576	2.27	441.1
abseil_btree	ycsb_b	4,194,304	1.64	610.9
abseil_btree	ycsb_b	16,777,216	1.09	917.8
libart	mixed	65,536	3.55	281.4
libart	mixed	262,144	3.65	273.8
libart	mixed	1,048,576	3.43	291.1
libart	mixed	4,194,304	2.60	384.8
libart	mixed	16,777,216	2.27	440.6
libart	rand_delete	65,536	6.51	153.6
libart	rand_delete	262,144	4.80	208.5
libart	rand_delete	1,048,576	3.38	295.8
libart	rand_delete	4,194,304	2.56	391.2
libart	rand_delete	16,777,216	1.94	515.0
libart	rand_insert	65,536	5.28	189.2
libart	rand_insert	262,144	4.25	235.2
libart	rand_insert	1,048,576	3.62	276.0
libart	rand_insert	4,194,304	2.83	352.9
libart	rand_insert	16,777,216	2.33	429.0

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
libart	search_after_churn	65,536	12.08	82.8
libart	search_after_churn	262,144	7.65	130.7
libart	search_after_churn	1,048,576	7.18	139.2
libart	search_after_churn	4,194,304	4.28	233.7
libart	search_after_churn	16,777,216	4.03	248.1
libart	seq_insert	65,536	5.27	189.9
libart	seq_insert	262,144	5.89	169.9
libart	seq_insert	1,048,576	6.93	144.3
libart	seq_insert	4,194,304	8.30	120.5
libart	seq_insert	16,777,216	8.21	121.8
libart	ycsb_a	65,536	4.58	218.4
libart	ycsb_a	262,144	5.81	172.1
libart	ycsb_a	1,048,576	6.26	159.7
libart	ycsb_a	4,194,304	5.96	167.7
libart	ycsb_a	16,777,216	5.59	178.8
libart	ycsb_b	65,536	7.69	130.1
libart	ycsb_b	262,144	5.46	183.1
libart	ycsb_b	1,048,576	5.01	199.5
libart	ycsb_b	4,194,304	3.96	252.6
libart	ycsb_b	16,777,216	3.25	308.1
matryoshka	mixed	65,536	4.86	205.6
matryoshka	mixed	262,144	4.54	220.4
matryoshka	mixed	1,048,576	4.15	241.0
matryoshka	mixed	4,194,304	3.03	329.6
matryoshka	mixed	16,777,216	2.17	460.7
matryoshka	rand_delete	65,536	3.87	258.3
matryoshka	rand_delete	262,144	2.98	335.8
matryoshka	rand_delete	1,048,576	2.46	406.5
matryoshka	rand_delete	4,194,304	2.01	496.9
matryoshka	rand_delete	16,777,216	1.28	778.6
matryoshka	rand_insert	65,536	4.18	239.3
matryoshka	rand_insert	262,144	3.37	297.0
matryoshka	rand_insert	1,048,576	2.75	363.8
matryoshka	rand_insert	4,194,304	2.05	487.9
matryoshka	rand_insert	16,777,216	1.36	734.7
matryoshka	search_after_churn	65,536	5.65	177.0
matryoshka	search_after_churn	262,144	4.27	234.2
matryoshka	search_after_churn	1,048,576	3.23	309.4
matryoshka	search_after_churn	4,194,304	2.23	448.4
matryoshka	search_after_churn	16,777,216	1.73	578.0
matryoshka	seq_insert	65,536	11.73	85.3
matryoshka	seq_insert	262,144	11.24	89.0
matryoshka	seq_insert	1,048,576	10.70	93.5
matryoshka	seq_insert	4,194,304	9.71	103.0
matryoshka	seq_insert	16,777,216	8.83	113.3
matryoshka	ycsb_a	65,536	10.34	96.8

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
matryoshka	ycsb_a	262,144	9.03	110.8
matryoshka	ycsb_a	1,048,576	7.61	131.4
matryoshka	ycsb_a	4,194,304	7.01	142.7
matryoshka	ycsb_a	16,777,216	5.79	172.6
matryoshka	ycsb_b	65,536	4.77	209.6
matryoshka	ycsb_b	262,144	4.35	230.1
matryoshka	ycsb_b	1,048,576	3.14	318.1
matryoshka	ycsb_b	4,194,304	2.25	443.6
matryoshka	ycsb_b	16,777,216	1.55	644.6
std_set	mixed	65,536	2.18	458.4
std_set	mixed	262,144	1.78	560.5
std_set	mixed	1,048,576	0.88	1,138.3
std_set	mixed	4,194,304	0.67	1,502.3
std_set	mixed	16,777,216	0.58	1,714.1
std_set	rand_delete	65,536	1.13	886.6
std_set	rand_delete	262,144	0.83	1,205.1
std_set	rand_delete	1,048,576	0.58	1,738.3
std_set	rand_delete	4,194,304	0.42	2,363.8
std_set	rand_delete	16,777,216	0.32	3,091.9
std_set	rand_insert	65,536	2.58	387.1
std_set	rand_insert	262,144	1.99	503.4
std_set	rand_insert	1,048,576	0.97	1,025.8
std_set	rand_insert	4,194,304	0.57	1,764.9
std_set	rand_insert	16,777,216	0.37	2,672.1
std_set	search_after_churn	65,536	2.34	427.8
std_set	search_after_churn	262,144	1.09	920.5
std_set	search_after_churn	1,048,576	0.54	1,853.2
std_set	search_after_churn	4,194,304	0.33	3,076.5
std_set	search_after_churn	16,777,216	0.26	3,829.6
std_set	seq_insert	65,536	4.13	242.0
std_set	seq_insert	262,144	3.08	324.9
std_set	seq_insert	1,048,576	2.20	454.6
std_set	seq_insert	4,194,304	1.58	633.6
std_set	seq_insert	16,777,216	1.52	659.0
std_set	ycsb_a	65,536	2.57	389.7
std_set	ycsb_a	262,144	2.21	452.1
std_set	ycsb_a	1,048,576	1.79	559.6
std_set	ycsb_a	4,194,304	1.15	868.9
std_set	ycsb_a	16,777,216	1.31	763.7
std_set	ycsb_b	65,536	2.09	479.6
std_set	ycsb_b	262,144	1.02	983.8
std_set	ycsb_b	1,048,576	0.60	1,663.4
std_set	ycsb_b	4,194,304	0.40	2,498.2
std_set	ycsb_b	16,777,216	0.31	3,178.8
tlx_btree	mixed	65,536	3.24	308.9
tlx_btree	mixed	262,144	3.77	265.4

Continued on next page

Table 4: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
tlx_btree	mixed	1,048,576	2.60	383.9
tlx_btree	mixed	4,194,304	1.76	569.5
tlx_btree	mixed	16,777,216	1.36	735.3
tlx_btree	rand_delete	65,536	2.91	344.0
tlx_btree	rand_delete	262,144	2.70	370.3
tlx_btree	rand_delete	1,048,576	1.72	580.4
tlx_btree	rand_delete	4,194,304	1.22	818.1
tlx_btree	rand_delete	16,777,216	0.87	1,143.0
tlx_btree	rand_insert	65,536	2.79	358.8
tlx_btree	rand_insert	262,144	2.16	462.9
tlx_btree	rand_insert	1,048,576	2.14	467.8
tlx_btree	rand_insert	4,194,304	1.40	715.5
tlx_btree	rand_insert	16,777,216	0.98	1,019.1
tlx_btree	search_after_churn	65,536	3.40	294.3
tlx_btree	search_after_churn	262,144	2.41	415.4
tlx_btree	search_after_churn	1,048,576	1.63	612.5
tlx_btree	search_after_churn	4,194,304	1.09	918.8
tlx_btree	search_after_churn	16,777,216	0.84	1,184.1
tlx_btree	seq_insert	65,536	8.86	112.8
tlx_btree	seq_insert	262,144	7.96	125.7
tlx_btree	seq_insert	1,048,576	8.94	111.8
tlx_btree	seq_insert	4,194,304	7.87	127.1
tlx_btree	seq_insert	16,777,216	6.99	143.1
tlx_btree	ycsb_a	65,536	8.35	119.8
tlx_btree	ycsb_a	262,144	7.65	130.6
tlx_btree	ycsb_a	1,048,576	7.32	136.5
tlx_btree	ycsb_a	4,194,304	5.94	168.4
tlx_btree	ycsb_a	16,777,216	5.10	196.0
tlx_btree	ycsb_b	65,536	2.69	371.5
tlx_btree	ycsb_b	262,144	2.54	394.4
tlx_btree	ycsb_b	1,048,576	1.71	585.8
tlx_btree	ycsb_b	4,194,304	1.06	946.9
tlx_btree	ycsb_b	16,777,216	0.81	1,239.4

10 Analysis and Diagnosis

10.1 Nested Sub-Tree Modification Cost

Each insert or delete navigates the page-level CL sub-tree (2–3 SIMD comparisons on 12–15 keys per level) to a target CL leaf, then performs a scalar `memmove` of at most 14 keys within that 64 B cache-line sub-node. The cost per modification is $O(h_s \times b)$ where $h_s \leq 2$ is the sub-tree height and $b = 15$ is the CL leaf capacity — roughly 30–45 key touches.

CL sub-node splits and merges occur only when a CL leaf overflows (15 keys) or underflows (< 7 keys). Page-level splits (extracting all keys and dividing across two pages) occur only when all 63 CL slots are exhausted. These are standard B+ tree structural operations, not full-page rebuilds.

Matryoshka achieves 2.75 Mop/s on random insert (N=1,048,576), vs. 0.97 Mop/s (`std::set`) and 3.62 Mop/s (fastest B-tree competitor).

10.2 SIMD at Every Level

SIMD `_mm_cmpgt_epi32`/`_mm_movemask_ps` is used for search within CL leaves (15 keys), CL internal nodes (12 separator keys), and outer internal nodes ($\leq$ 339 keys). Unlike a flat SIMD-blocked layout, the nested design retains SIMD benefits for both search *and* the navigation phase of insert/delete.

Search-after-churn throughput is 3.23 Mop/s at N=1,048,576.

10.3 Arena Allocator and TLB Effects

The arena allocator places leaves in 2 MiB hugepage-backed regions via `mmap(MAP_HUGETLB)`, falling back to `posix_memalign` + `madvise(MADV_HUGEPAGE)`:

- **Reduced dTLB misses**: one 2 MiB hugepage covers 512 leaf pages. dTLB rate: N/A/1,000 ops vs. N/A for `std::set`.
- **Spatial locality**: co-located leaves benefit sequential scans and range iteration.

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). Its $O(\log N)$ insert involves a constant-factor pointer update but pays a cache miss at every tree level.

10.5 Where TLX and Abseil Compete

Both use sorted-array B-tree leaves with `memmove` insert ($B \approx 64$–256). They stay within 7% of each other. Matryoshka is competitive on insert-heavy workloads thanks to the nested sub-tree design, though sorted-array leaves have a simpler constant factor for small `memmove` operations. Neither TLX nor Abseil uses SIMD for in-leaf search.

10.6 ART's Radix Approach

ART performs O(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 Assessment

The matryoshka nesting design achieves competitive insert and delete throughput while preserving SIMD-accelerated search at every level of the hierarchy. Each modification touches a single cache-line sub-node rather than rebuilding an entire page, yielding throughput within 2× of the best B-tree competitor on insert-heavy workloads and 2× on delete-heavy workloads at the largest dataset size (N=16,777,216).

11 Future Directions

11.1 Deeper Nesting: Superpage-Level Sub-Trees

The current design nests CL sub-nodes within 4 KiB pages. A natural extension is to nest 4 KiB page sub-nodes within 2 MiB superpages, adding a third level to the matryoshka hierarchy. This would reduce outer-tree height and confine more operations within a single TLB entry. `mt_hierarchy_init_superpage` provides the skeleton for this.

11.2 Wider SIMD: AVX2 and AVX-512

The current CL sub-nodes are sized for SSE2 (128-bit). AVX2 (256-bit) could double the keys per SIMD comparison, reducing sub-tree height. AVX-512 would enable 512-bit (one full cache line) operations, though the CL sub-node format would need restructuring.

11.3 Batch Insert API

Provide `matryoshka_insert_batch(tree, keys, k)`: sort incoming keys, route each to its target CL leaf, and batch the insertions to amortise sub-tree navigation. Useful for bulk-loading from unsorted input without pre-sorting.

11.4 Variable-Length Keys

The current 4-byte `int32_t` key format could be extended to variable-length keys by storing key offsets or using indirection within CL sub-nodes. This would broaden applicability at the cost of some cache efficiency.

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.