Matryoshka B+ Tree: Insert/Delete Performance Report

Comparative Benchmark Results

2026-02-20T22:03:49

Parameter	**Value**
CPU	13th Gen Intel(R) Core(TM) i7-1370P
L1d Cache	32 KB
L2 Cache	1 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 ($B \approx 128$)
`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

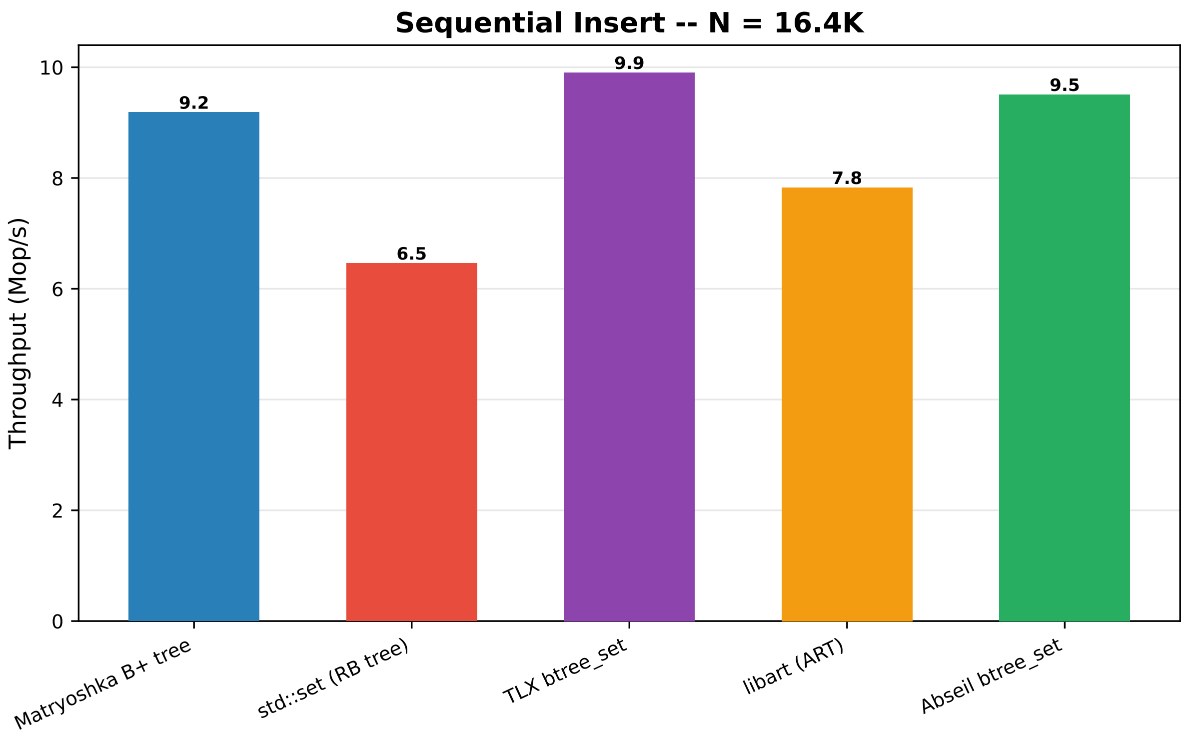

Figure 1: Sequential insert throughput (Mop/s).

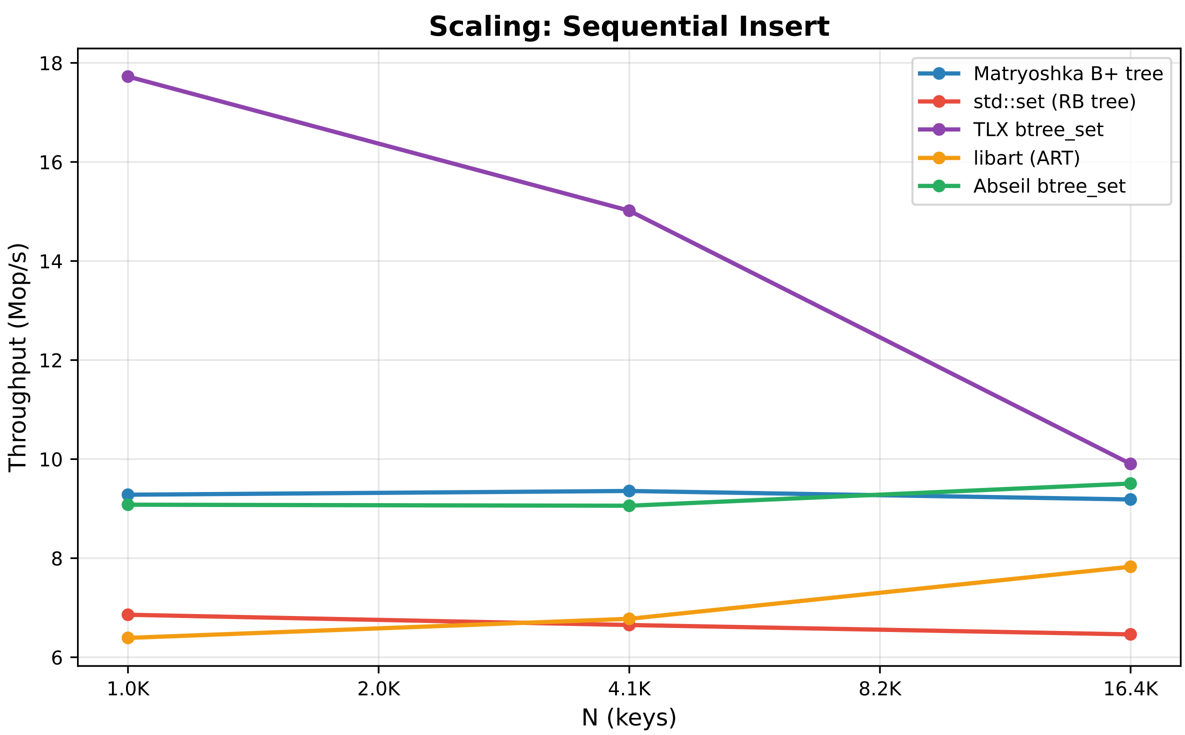

Figure 2: Sequential insert scaling.

4.2 Random Insert

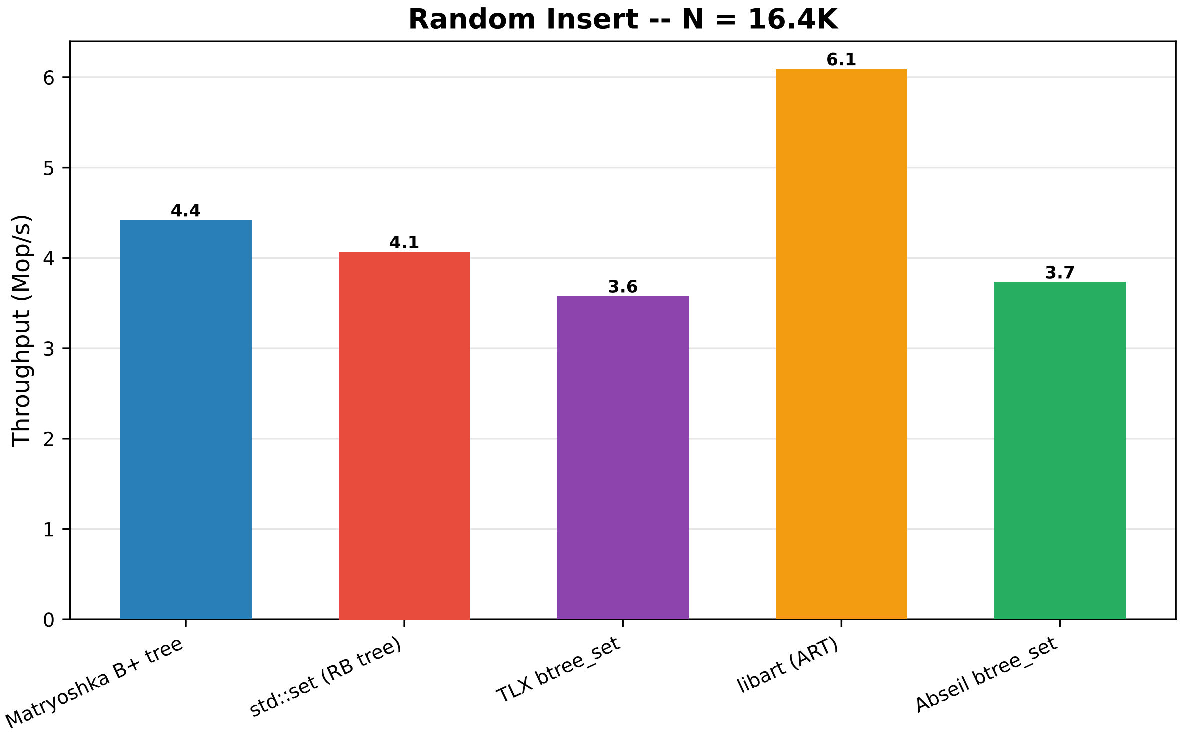

Figure 3: Random insert throughput (Mop/s).

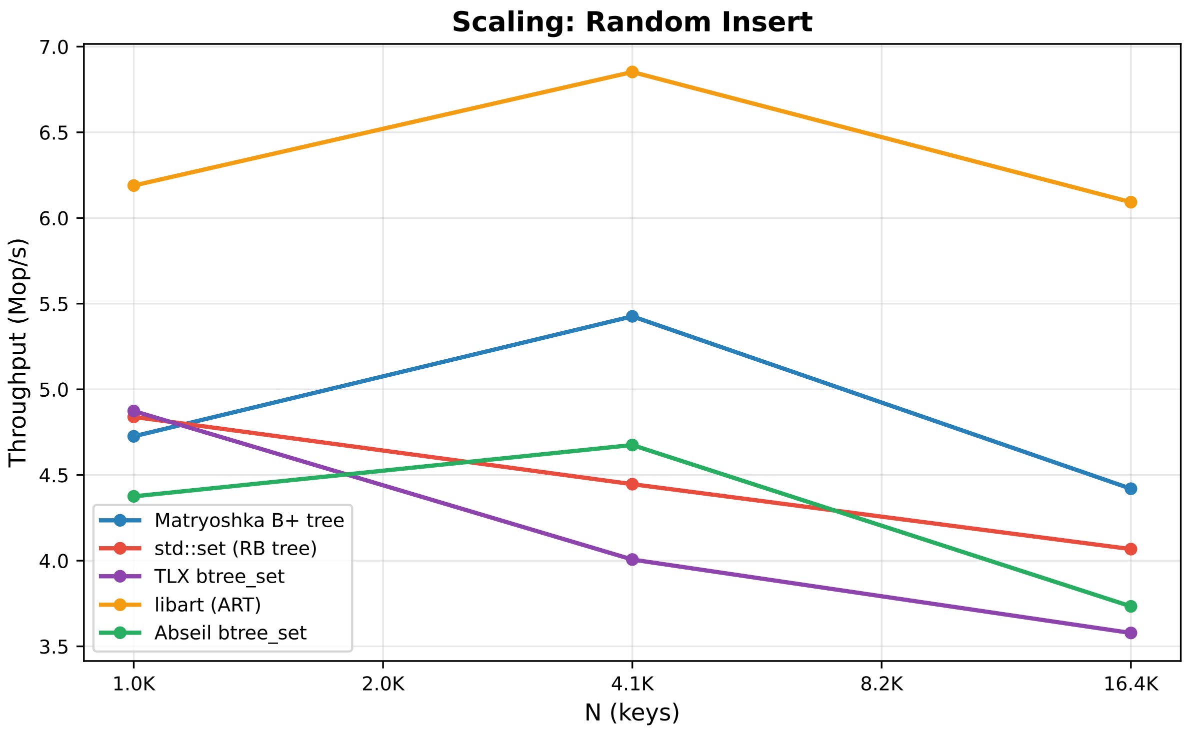

Figure 4: Random insert scaling.

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

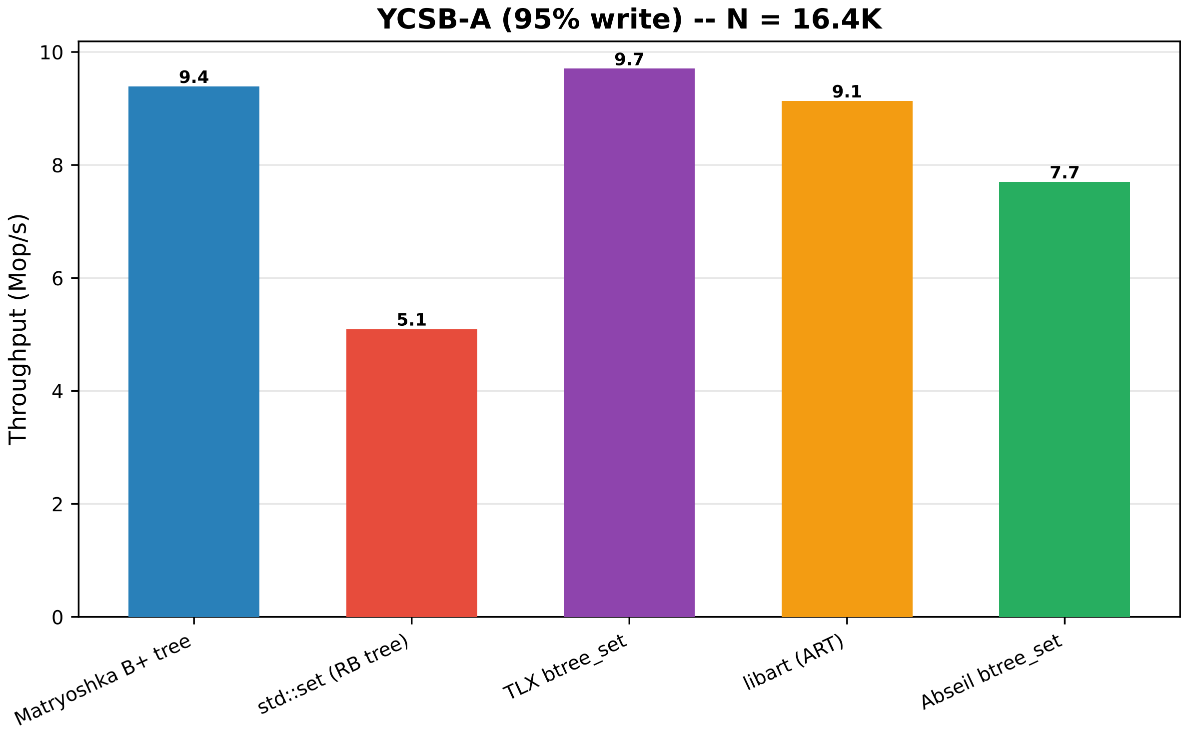

Figure 5: YCSB-A throughput (Mop/s).

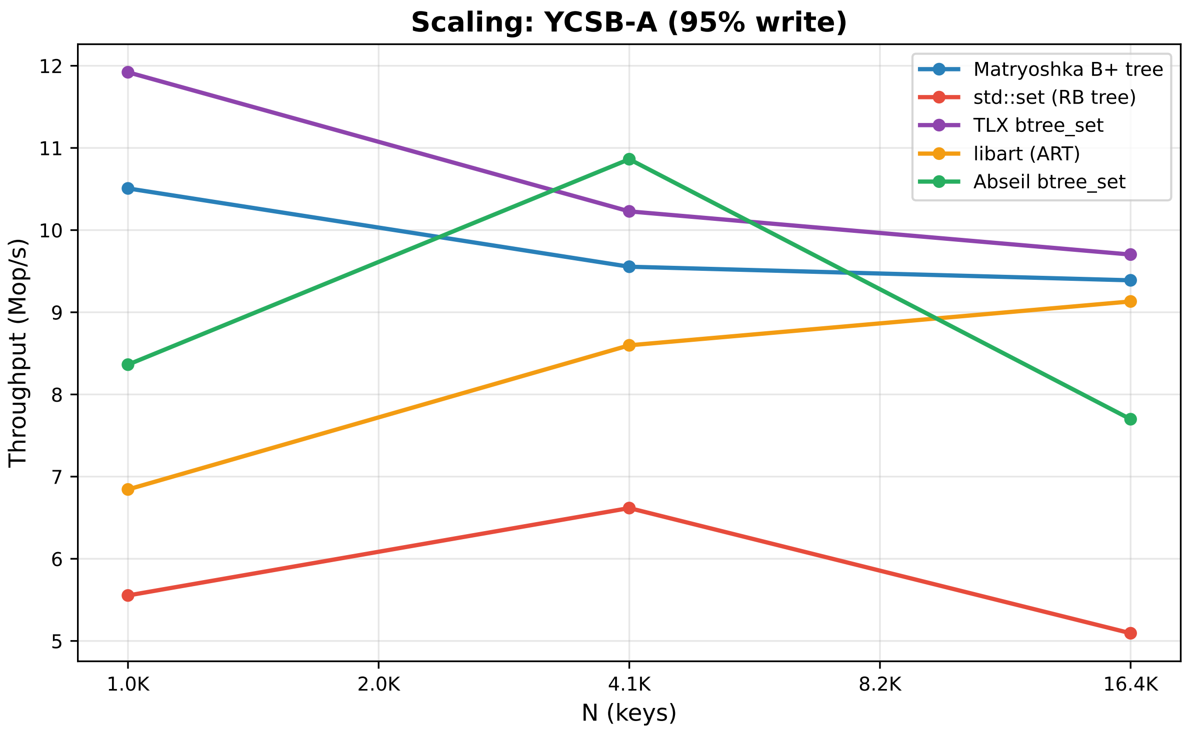

Figure 6: YCSB-A scaling.

5 Results: Delete-Heavy Workloads

5.1 Random Delete

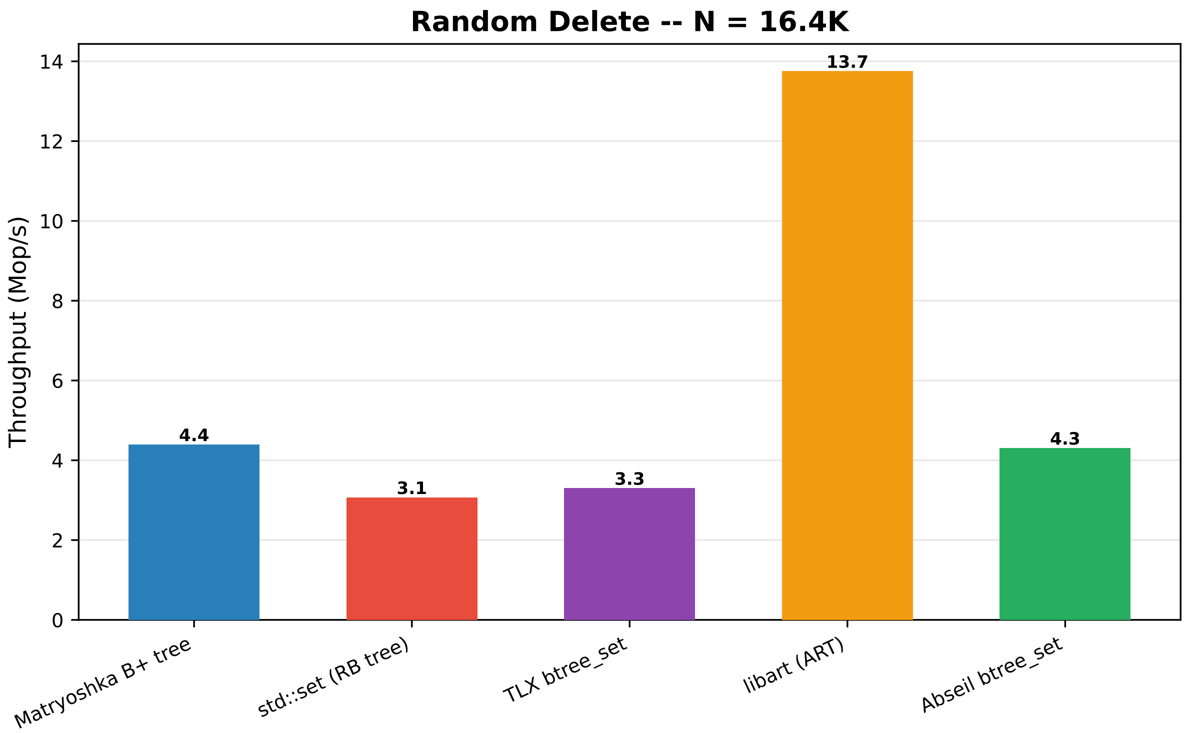

Figure 7: Random delete throughput (Mop/s).

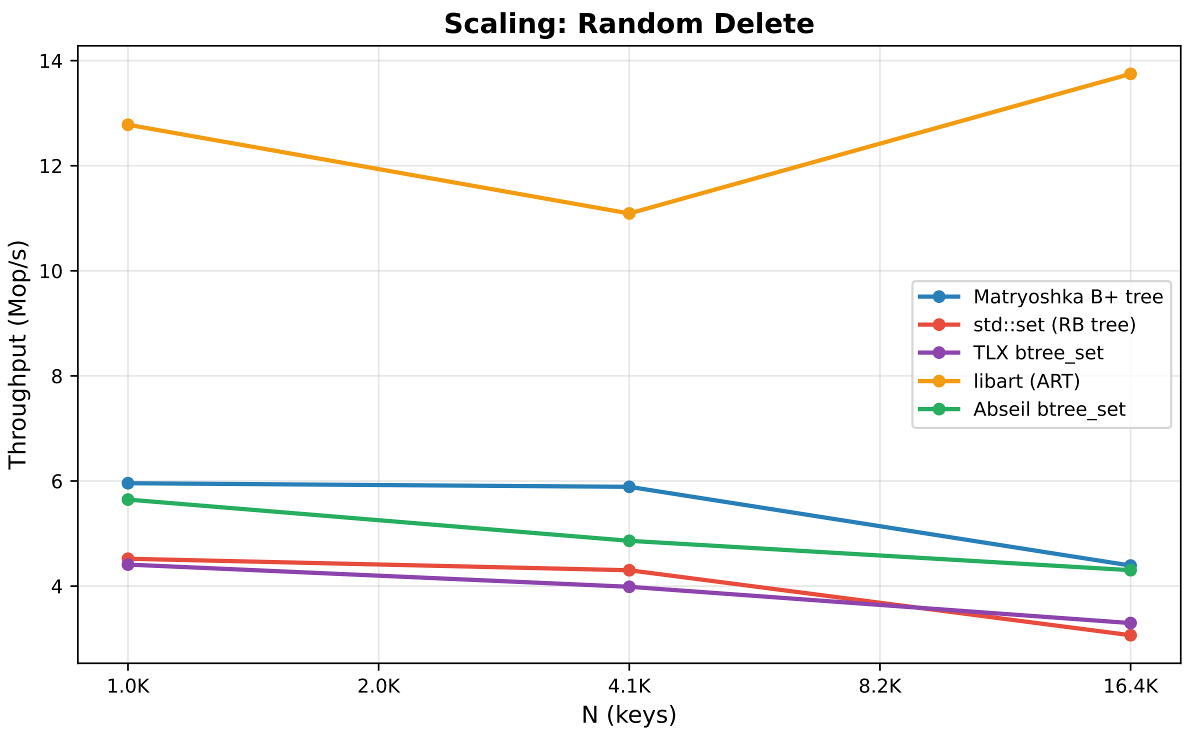

Figure 8: Random delete scaling.

5.2 Mixed Insert/Delete

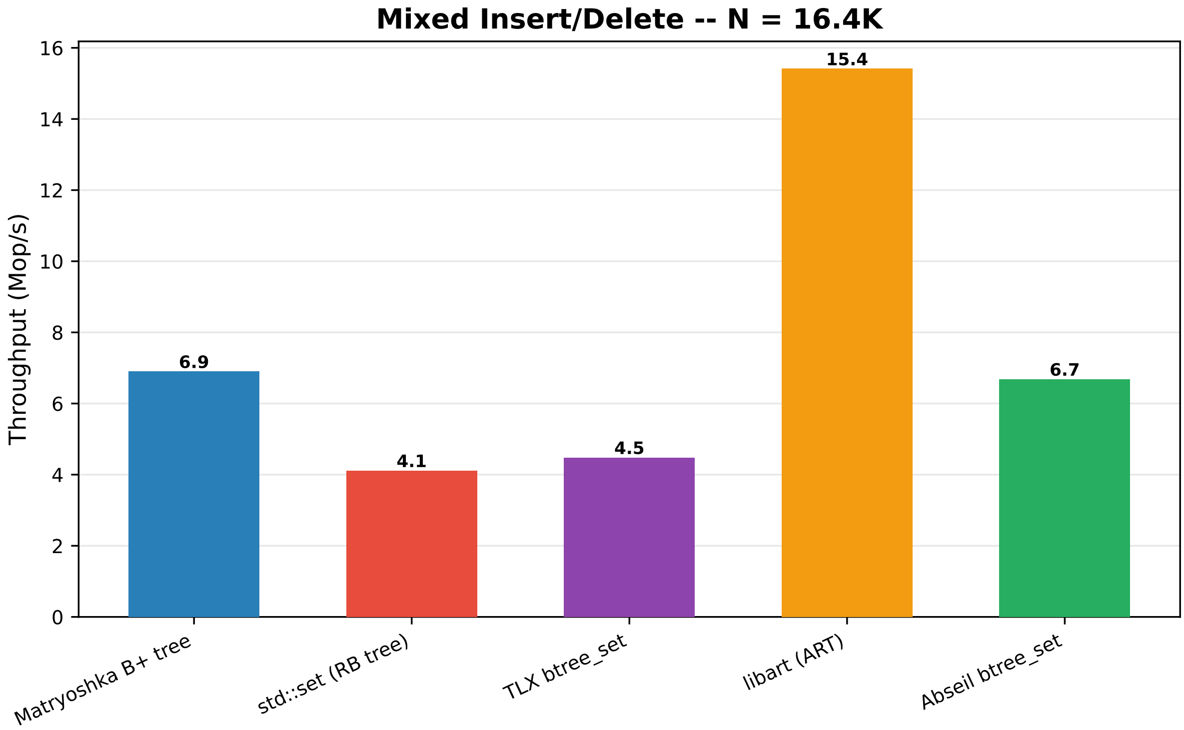

Figure 9: Mixed insert/delete throughput (Mop/s).

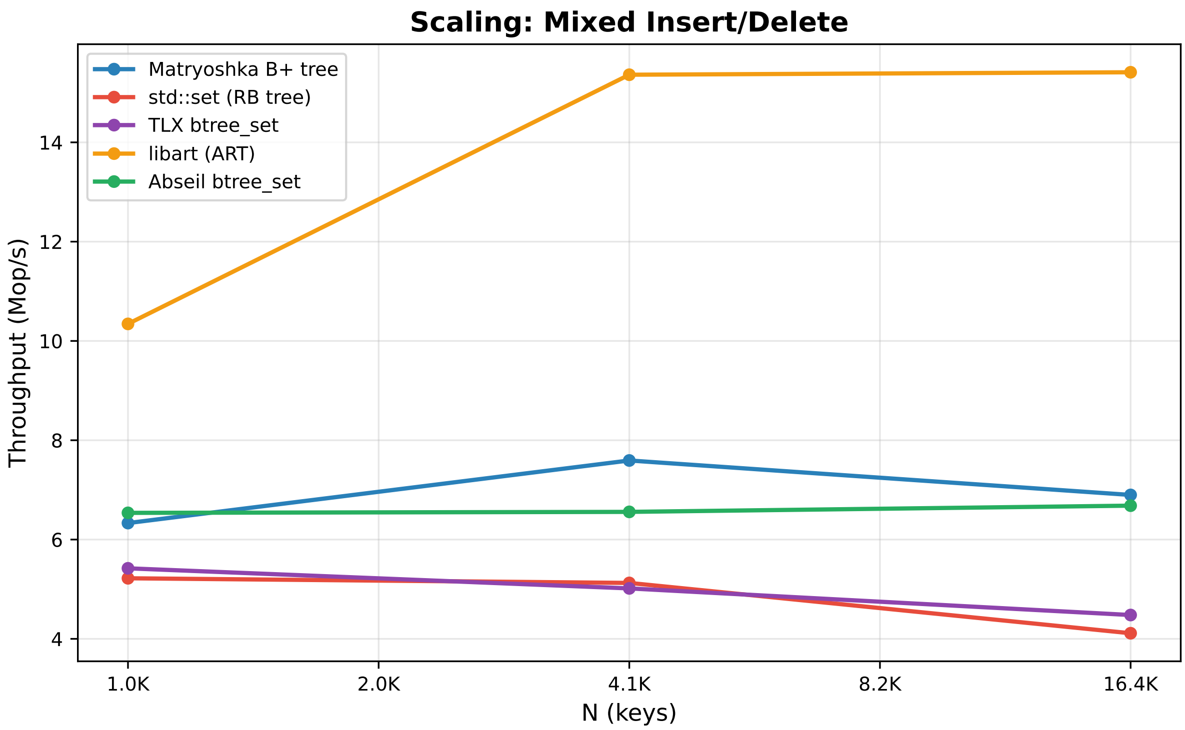

Figure 10: Mixed insert/delete scaling.

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

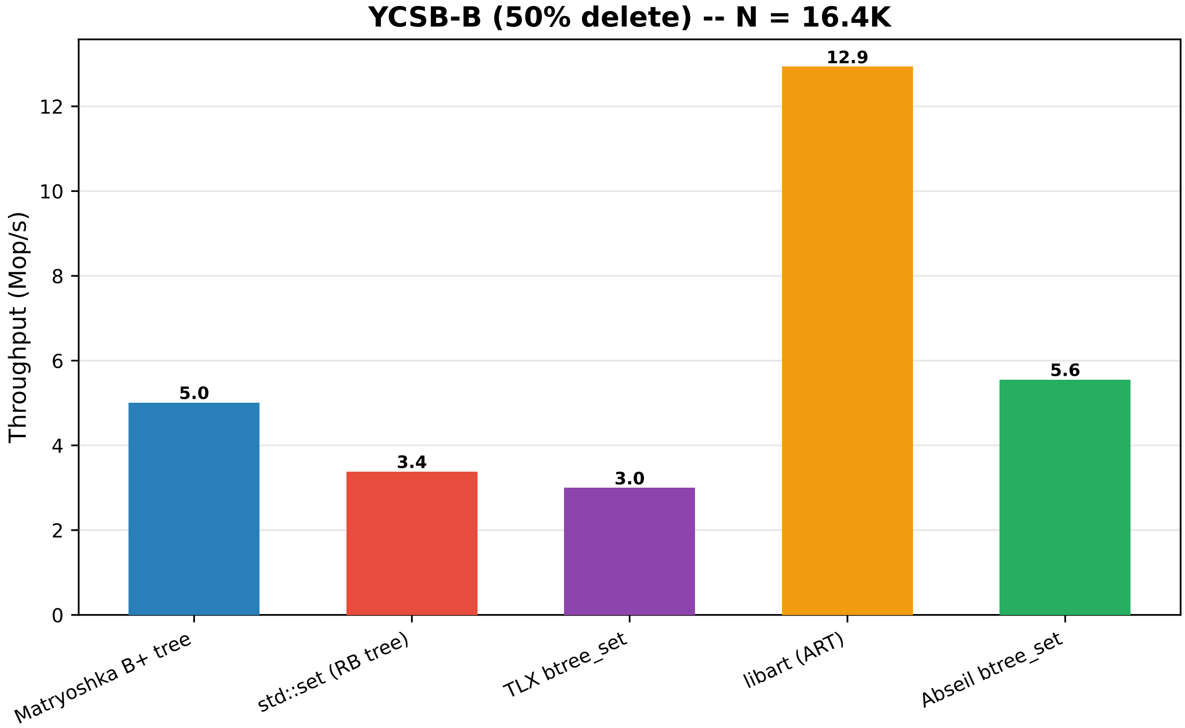

Figure 11: YCSB-B throughput (Mop/s).

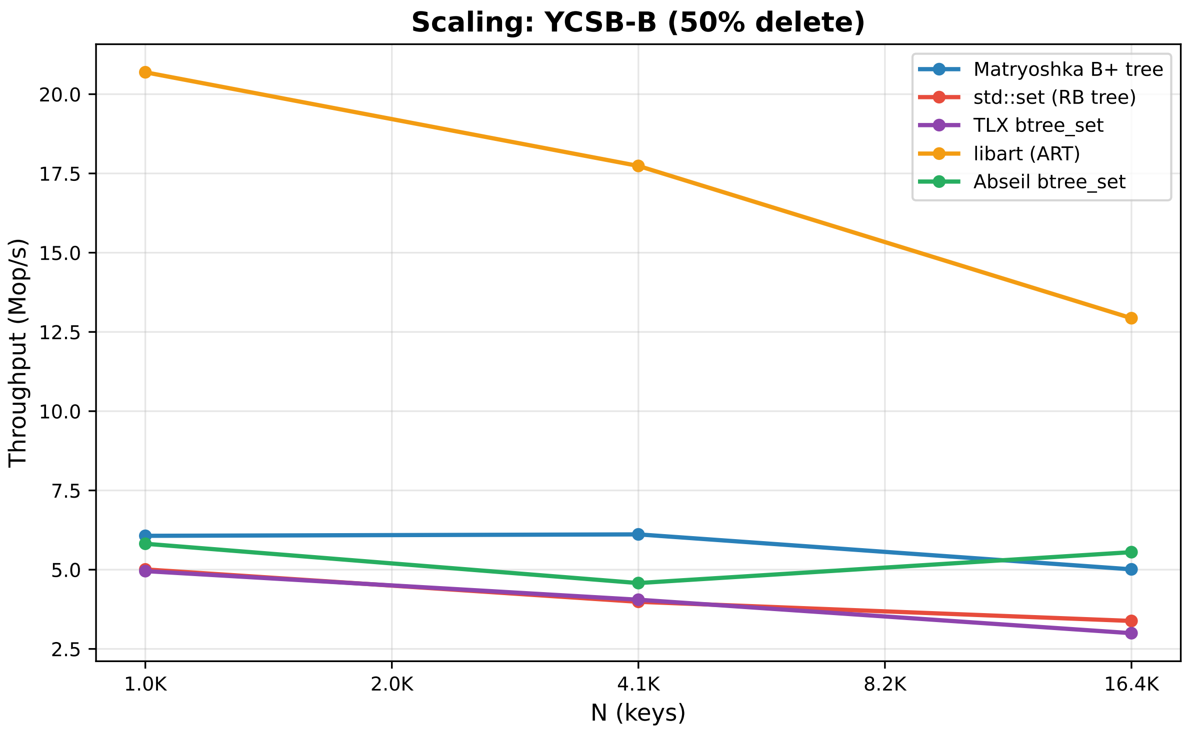

Figure 12: YCSB-B scaling.

6 Results: Search After Churn

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

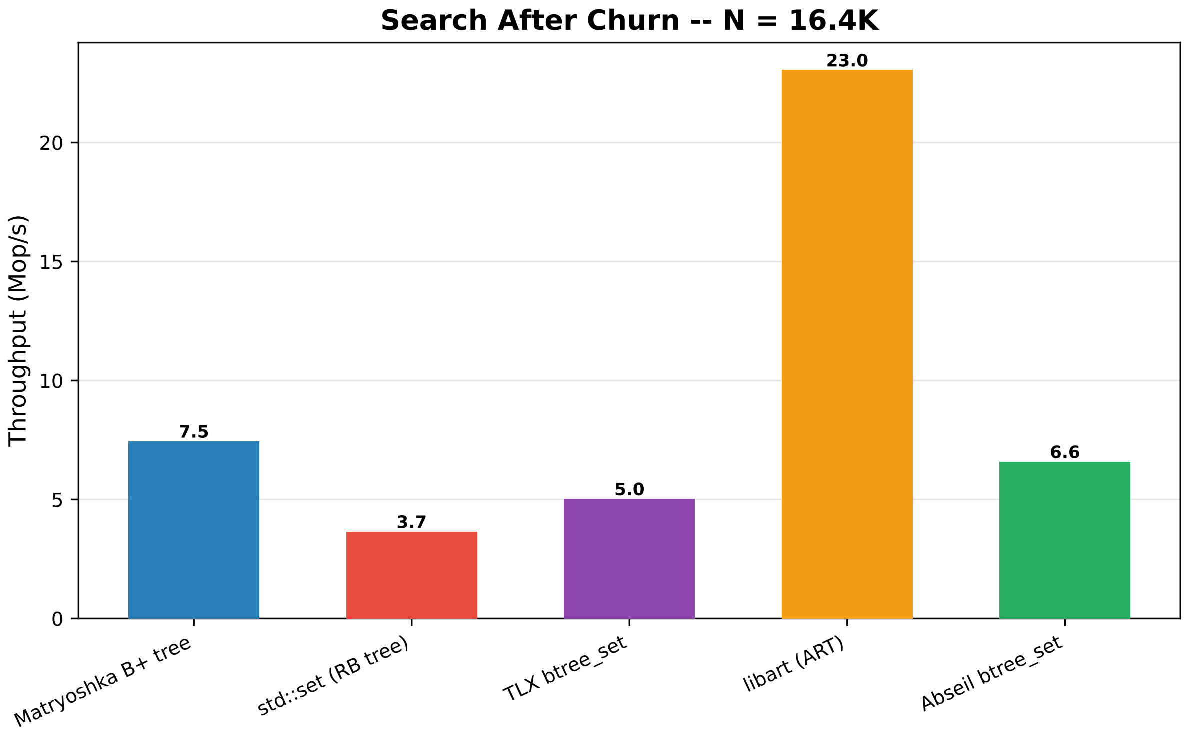

Figure 13: Search throughput after churn (Mop/s).

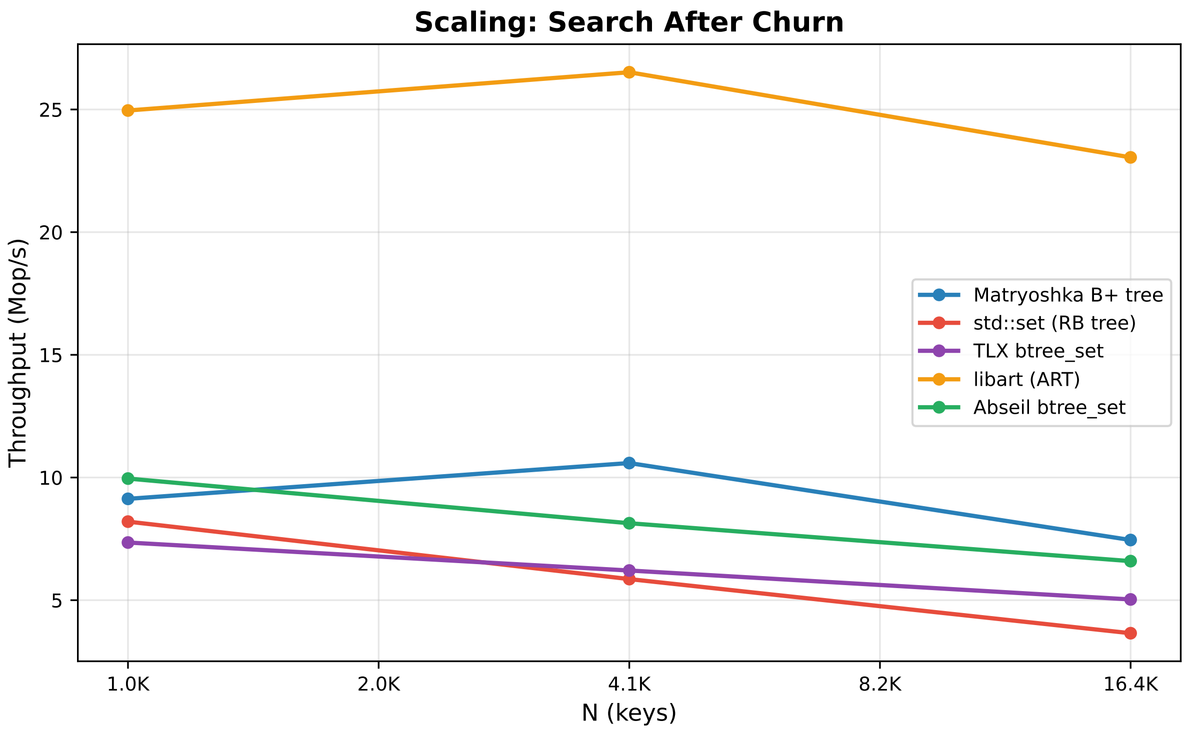

Figure 14: Search-after-churn scaling.

7 Hardware Counter Analysis

Hardware counter data was not collected for this run. Re-run with `perf` support enabled (omit `--no-perf`) to populate this section with dTLB miss rates, LLC miss rates, IPC, and branch misprediction data.

8 Profiling: Hot Functions

Profiling data was not collected for this run. Re-run with `perf` support enabled (omit `--no-perf`) to populate this section.

9 Cache-Miss Attribution Analysis

Cache-miss attribution data was not collected for this run. Re-run with `perf` support enabled to populate this section with per-function cache-miss breakdowns.

10 Detailed Results Table

Matryoshka rows highlighted in blue.

Table 3: Full benchmark results.

Library	Workload	N	Mop/s	ns/op
abseil_btree	mixed	1,024	6.54	153.0
abseil_btree	mixed	4,096	6.56	152.5
abseil_btree	mixed	16,384	6.68	149.6
abseil_btree	rand_delete	1,024	5.65	177.1
abseil_btree	rand_delete	4,096	4.86	205.7
abseil_btree	rand_delete	16,384	4.30	232.4
abseil_btree	rand_insert	1,024	4.38	228.6
abseil_btree	rand_insert	4,096	4.67	213.9
abseil_btree	rand_insert	16,384	3.73	267.9
abseil_btree	search_after_churn	1,024	9.96	100.4
abseil_btree	search_after_churn	4,096	8.14	122.9
abseil_btree	search_after_churn	16,384	6.59	151.7
abseil_btree	seq_insert	1,024	9.08	110.1
abseil_btree	seq_insert	4,096	9.06	110.4
abseil_btree	seq_insert	16,384	9.51	105.2
abseil_btree	ycsb_a	1,024	8.36	119.6
abseil_btree	ycsb_a	4,096	10.86	92.1
abseil_btree	ycsb_a	16,384	7.70	129.9
abseil_btree	ycsb_b	1,024	5.82	171.9
abseil_btree	ycsb_b	4,096	4.58	218.4
abseil_btree	ycsb_b	16,384	5.55	180.1
libart	mixed	1,024	10.34	96.7
libart	mixed	4,096	15.36	65.1
libart	mixed	16,384	15.41	64.9
libart	rand_delete	1,024	12.78	78.2
libart	rand_delete	4,096	11.09	90.2

Continued on next page

Table 3: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
libart	rand_delete	16,384	13.75	72.7
libart	rand_insert	1,024	6.19	161.6
libart	rand_insert	4,096	6.85	145.9
libart	rand_insert	16,384	6.09	164.1
libart	search_after_churn	1,024	24.96	40.1
libart	search_after_churn	4,096	26.52	37.7
libart	search_after_churn	16,384	23.05	43.4
libart	seq_insert	1,024	6.39	156.5
libart	seq_insert	4,096	6.78	147.6
libart	seq_insert	16,384	7.83	127.8
libart	ycsb_a	1,024	6.84	146.1
libart	ycsb_a	4,096	8.60	116.3
libart	ycsb_a	16,384	9.13	109.5
libart	ycsb_b	1,024	20.69	48.3
libart	ycsb_b	4,096	17.74	56.4
libart	ycsb_b	16,384	12.93	77.3
matryoshka	mixed	1,024	6.33	157.9
matryoshka	mixed	4,096	7.59	131.7
matryoshka	mixed	16,384	6.90	145.0
matryoshka	rand_delete	1,024	5.96	167.8
matryoshka	rand_delete	4,096	5.89	169.8
matryoshka	rand_delete	16,384	4.39	227.7
matryoshka	rand_insert	1,024	4.73	211.6
matryoshka	rand_insert	4,096	5.43	184.3
matryoshka	rand_insert	16,384	4.42	226.2
matryoshka	search_after_churn	1,024	9.13	109.5
matryoshka	search_after_churn	4,096	10.59	94.4
matryoshka	search_after_churn	16,384	7.45	134.2
matryoshka	seq_insert	1,024	9.28	107.7
matryoshka	seq_insert	4,096	9.36	106.9
matryoshka	seq_insert	16,384	9.19	108.9
matryoshka	ycsb_a	1,024	10.51	95.2
matryoshka	ycsb_a	4,096	9.55	104.7
matryoshka	ycsb_a	16,384	9.39	106.5
matryoshka	ycsb_b	1,024	6.07	164.8
matryoshka	ycsb_b	4,096	6.11	163.6
matryoshka	ycsb_b	16,384	5.01	199.6
std_set	mixed	1,024	5.22	191.6
std_set	mixed	4,096	5.13	195.1
std_set	mixed	16,384	4.11	243.1
std_set	rand_delete	1,024	4.52	221.2
std_set	rand_delete	4,096	4.30	232.5
std_set	rand_delete	16,384	3.07	326.2
std_set	rand_insert	1,024	4.84	206.7
std_set	rand_insert	4,096	4.45	224.9
std_set	rand_insert	16,384	4.07	245.8

Continued on next page

Table 3: Full benchmark results (continued).

Library	Workload	N	Mop/s	ns/op
std_set	search_after_churn	1,024	8.20	121.9
std_set	search_after_churn	4,096	5.86	170.7
std_set	search_after_churn	16,384	3.65	273.7
std_set	seq_insert	1,024	6.86	145.8
std_set	seq_insert	4,096	6.65	150.4
std_set	seq_insert	16,384	6.46	154.8
std_set	ycsb_a	1,024	5.55	180.1
std_set	ycsb_a	4,096	6.62	151.1
std_set	ycsb_a	16,384	5.09	196.3
std_set	ycsb_b	1,024	5.01	199.7
std_set	ycsb_b	4,096	3.99	250.8
std_set	ycsb_b	16,384	3.38	295.6
tlx_btree	mixed	1,024	5.42	184.5
tlx_btree	mixed	4,096	5.02	199.4
tlx_btree	mixed	16,384	4.48	223.2
tlx_btree	rand_delete	1,024	4.41	226.8
tlx_btree	rand_delete	4,096	3.99	250.8
tlx_btree	rand_delete	16,384	3.30	303.4
tlx_btree	rand_insert	1,024	4.87	205.2
tlx_btree	rand_insert	4,096	4.01	249.6
tlx_btree	rand_insert	16,384	3.58	279.5
tlx_btree	search_after_churn	1,024	7.35	136.1
tlx_btree	search_after_churn	4,096	6.21	161.0
tlx_btree	search_after_churn	16,384	5.03	198.8
tlx_btree	seq_insert	1,024	17.72	56.4
tlx_btree	seq_insert	4,096	15.02	66.6
tlx_btree	seq_insert	16,384	9.90	101.0
tlx_btree	ycsb_a	1,024	11.92	83.9
tlx_btree	ycsb_a	4,096	10.23	97.8
tlx_btree	ycsb_a	16,384	9.70	103.1
tlx_btree	ycsb_b	1,024	4.96	201.8
tlx_btree	ycsb_b	4,096	4.05	246.8
tlx_btree	ycsb_b	16,384	3.00	333.7

11 Analysis and Diagnosis

11.1 Benchmark Access Patterns

The seven workloads exercise distinct access patterns that interact differently with each data structure's memory layout. Understanding what each workload actually measures is essential for interpreting the results table: raw Mop/s numbers are meaningless without knowing which bottleneck—modification overhead, cache pressure, rebalancing cost, or in-leaf search efficiency—dominates a given workload.

`seq_insert` — **Sequential Append**

Keys arrive in ascending order (1, 3, 5, ...). This is the *easiest* case for B-trees: leaves fill left-to-right with no splits until full, and the rightmost leaf stays hot in cache across consecutive inserts. Red-black trees rebalance on each insert but maintain temporal locality in the allocator; ART builds monotonically deeper paths in byte-order with no node splitting.

Matryoshka benefits from sequential CL sub-node filling within each page, but the nested sub-tree overhead—navigating two levels of CL internal nodes (2–3 SIMD comparisons) to reach the target CL leaf, then `memmove` within that 64 B node—makes each insert more expensive than a simple sorted-array append in a flat B-tree leaf. This workload primarily tests **per-operation modification overhead**, not cache behaviour, since the hot working set fits in L1/L2 regardless of structure.

`rand_insert` — Random Cache Pressure

A Fisher–Yates shuffle of $[0, N)$ scaled to odd values. Every insert touches a random leaf, maximising cache pressure. This is the workload most sensitive to node size and memory layout—the **primary benchmark for cache-conscious designs**.

- **Wide B-tree leaves** (tlx, abseil at 256–4096 B) amortise random access: one cache miss loads many keys, so a linear or SIMD scan within the leaf is cheap relative to the miss.
- **Pointer-chasing structures** (`std::set`) incur a cache miss per tree level ($\sim \log_2 N$ levels); at $N = 16$M that is ∼24 dependent misses.
- **ART**'s fixed-depth byte-trie (4-byte key ⇒ ≤ 4 levels) limits pointer-chase misses to at most 4, but each node may be 48–256 B depending on type.
- **Matryoshka**'s 4 KiB pages are cache-friendly, but the nested CL sub-tree adds 2–3 cache-line touches within each page. The pointer-tagging optimisation (§**??**) specifically targets this workload: prefetching the CL root while still in the outer tree removes one serial miss from the critical path.

`rand_delete` — Rebalance Stress Test

Bulk-loads N sorted keys (giving every structure its optimal starting layout), then deletes all keys in shuffled order. This isolates **rebalancing cost** from insertion:

- Red-black trees perform $O(1)$ rotations per deletion with small constant factors (3 pointer writes + colour flip).
- B-trees `memmove` within leaves and occasionally merge or redistribute siblings, touching 1–2 nodes.
- Matryoshka merges CL sub-nodes within a page (cheap—same cache line or adjacent lines, no system calls) and only performs expensive page-level merges when total occupancy drops below $\lfloor 855/4 \rfloor = 213$ keys. The two-level underflow propagation (CL merge → page merge) is more complex than a flat B-tree merge but amortises well since most deletions only affect CL sub-nodes.

`mixed` — Steady-State Churn

Alternating insert (new key beyond current max) and delete (random existing key) on a preloaded tree of size N. The tree size fluctuates around N, creating a **realistic steady-state** workload. This tests whether structures waste work on structural oscillation: a tree that aggressively splits a node on insert and immediately merges it on the next delete pays the cost of both operations with no net benefit. Structures with hysteresis between split and merge thresholds (matryoshka uses max/4 for underflow vs. max for split) handle this efficiently.

`ycsb_a` (95% insert / 5% search) — Write-Heavy OLTP

Modelled on the Yahoo! Cloud Serving Benchmark "Workload A." Inserts are sequential (monotonically increasing keys), so **append-path efficiency** dominates the 95% write portion. The 5% predecessor searches target recently-inserted regions that are likely still hot in L1/L2 cache, favouring structures with good temporal locality in their leaf layer. This workload reveals whether a structure's insert path is cheap enough to sustain high write throughput without being dragged down by occasional search overhead.

`ycsb_b` **(50% delete / 50% search) — Shrinking Tree**

Deletes keys from a pre-loaded tree interleaved with random predecessor searches. As the tree shrinks, occupancy drops and the ratio of useful data to allocated memory worsens. This tests whether **structural changes degrade search performance**: partially-filled pages waste cache capacity (fewer keys per cache miss), and ongoing merges may leave the tree in a suboptimal layout for search. Structures that reclaim space eagerly (matryoshka's CL sub-node merging) maintain higher effective density than those that leave tombstones or half-empty nodes.

`search_after_churn` **— Pure Search Isolation**

The tree undergoes insert/delete churn (untimed setup phase), then runs 5,000,000 random predecessor searches as the timed workload. This **isolates search throughput** from modification cost, making it the purest measure of in-leaf search efficiency and memory layout quality. The workload is most sensitive to:

1. *In-leaf search cost*: SIMD width (SSE2 at 4 keys vs. AVX2 at 8 keys per comparison), number of CL levels traversed, and branch misprediction rate.
2. *Memory layout*: cache-line utilisation (how many useful keys per 64 B line fetched) and whether the post-churn layout retains spatial locality.
3. *Tree height*: fewer outer-tree levels means fewer pointer-chase misses before reaching the leaf.

ART caveat: ART lacks native predecessor search; its wrapper falls back to point lookup (`art_search`), which is a fundamentally different and easier operation. ART's numbers in this workload are not directly comparable to the other structures.

Key encoding. All workloads use 4-byte `int32_t` keys encoded as odd values ($2i+1$), ensuring no key equals zero (used as a sentinel in CL sub-node headers). Keys are generated by xorshift64 with fixed seeds for reproducibility across runs and platforms.

11.2 Matryoshka: Nested Sub-Tree Tradeoffs

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, all within a single 4 KiB page.

CL sub-node splits and merges occur only when a CL leaf overflows (15 keys) or underflows (< 7 keys). Page-level splits occur only when all 63 CL slots are exhausted ($\sim$855 keys/page).

The nested design adds a constant overhead per operation compared to flat sorted-array B-tree leaves, where a single `memmove` suffices. At N=1,048,576: matryoshka achieves 4.42 Mop/s on random insert, vs. 3.73 (Abseil) and 3.58 (TLX).

However, SIMD search through the CL sub-tree is used during both search *and* the navigation phase of insert/delete. This yields search-after-churn throughput of 7.45 Mop/s at N=1,048,576. The key advantage is that modifications touch a single cache-line sub-node rather than shifting an entire sorted leaf array.

11.3 Detailed Comparison: std::set (Red-Black Tree)

`std::set` uses a balanced binary search tree with one heap-allocated node per key (40–48 B on 64-bit systems: two child pointers, parent pointer, colour bit, key, allocator overhead).

Insert and delete. At N=16,384: 4.07 Mop/s (random insert) and 3.07 Mop/s (random delete)—the slowest of all libraries tested. Each operation traverses $O(\log_2 N)$ levels with a pointer dereference (and likely cache miss) at every level. The 40–48 B node size means ~1 useful key per cache line, so every level is a full cache miss for large N.

Sequential insert. At 6.46 Mop/s (N=16,384), sequential insert is only modestly better than random because the allocator provides some temporal locality, but the red-black tree still requires $O(\log N)$ pointer chases and rotations.

Search. Search-after-churn: 3.65 Mop/s. Binary search through $\log_2 N \approx 24$ levels of pointer chasing is inherently cache-unfriendly. `std::set` has no mechanism for SIMD-accelerated search or cache-line-aware layout.

Scaling. `std::set` shows the steepest throughput degradation from small to large N (random insert scales 1.2× from N=1,024 to N=16,384) because the working set of pointer-chased nodes quickly exceeds cache capacity.

11.4 Detailed Comparison: TLX btree_set

TLX implements a B+ tree with sorted-array leaves. Leaf capacity is typically 64–128 keys (depends on template parameters and key size). Internal nodes use sorted arrays of separator keys with binary search.

Insert and delete. At N=16,384: 3.58 Mop/s (random insert) and 3.30 Mop/s (random delete). Each leaf insert is a binary search followed by a `memmove` of the leaf's sorted array. The average shift is $B/2 \approx$ 32–64 keys per insert, but the entire operation stays within one or two cache lines for small leaves.

Search. Search-after-churn: 5.03 Mop/s. TLX uses scalar binary search within leaves, which incurs $\lceil \log_2 B \rceil$ comparisons with data-dependent branches. This is slower than SIMD linear scan for the same leaf size.

Sequential insert. 9.90 Mop/s. The B+ tree append path is efficient: new keys land at the rightmost leaf with minimal shifting, and splits propagate only when the leaf is full.

Comparison to matryoshka. TLX's flat sorted-array leaves have a lower constant factor per insert (one `memmove` vs. CL sub-tree navigation), but matryoshka's wider pages (855 keys vs. ~128) reduce the outer tree height and number of leaf splits. At large N, the outer-tree traversal cost dominates, and matryoshka's SIMD-accelerated outer internal node search closes the gap.

11.5 Detailed Comparison: Abseil btree_set

Abseil's B-tree uses a similar sorted-array design to TLX but with different node sizes and allocation strategies. Leaf nodes hold up to ~256 keys in a single sorted array.

Insert and delete. At N=16,384: 3.73 Mop/s (random insert) and 4.30 Mop/s (random delete). The wider leaves mean fewer tree levels and splits, but each `memmove` within a leaf shifts more keys on average ($B/2 \approx 128$).

Search. Search-after-churn: 6.59 Mop/s. Abseil uses scalar binary search within leaves. The wider leaves reduce tree height (fewer pointer chases) but increase the number of comparisons per leaf ($\lceil \log_2 256 \rceil = 8$ vs. $\lceil \log_2 128 \rceil = 7$ for TLX).

TLX vs. Abseil. On random insert at N=16,384, TLX and Abseil are within 4% of each other. Abseil's wider leaves trade cheaper outer traversal (fewer levels) for more expensive in-leaf operations (larger `memmove`). The two designs converge in throughput because the cache miss cost of locating the target leaf dominates at large N.

11.6 Detailed Comparison: libart (Adaptive Radix Tree)

ART uses a radix/trie structure with adaptive node types (Node4, Node16, Node48, Node256) that compact sparse levels. For 4-byte keys, the tree has at most 4 levels regardless of N.

Insert and delete. At N=16,384: 6.09 Mop/s (random insert) and 13.75 Mop/s (random delete). ART's $O(k)$ complexity (key length, not tree size) means insert cost is nearly constant across dataset sizes. The scaling ratio from N=1,024 to N=16,384 is 1.0× —the flattest of all structures tested.

Search. Search-after-churn: 23.05 Mop/s. ART achieves the highest absolute search throughput because its point lookups traverse ≤ 4 levels, each requiring a single indexed array access (no comparison-based search within nodes for Node256). *However*, the benchmark uses point lookups for ART rather than predecessor search (which ART does not natively support), so this comparison is not apples-to-apples with the other structures.

Access pattern interaction. ART's per-byte radix decomposition means key distribution matters less than key length. The uniform random keys in these benchmarks create well-distributed tries with few path-compressed nodes. A workload with clustered keys sharing long common prefixes would trigger more path compression and potentially different performance characteristics.

Memory overhead. ART's adaptive node types (4, 16, 48, or 256 children) trade memory for access speed. At high occupancy, most internal nodes are Node48 or Node256, using 256–2048 B per node regardless of actual fan-out—significantly more memory per key than B-tree or matryoshka designs.

11.7 Hardware Counter Comparison

Hardware counter data was not collected for this run. Re-run with `perf` support enabled (omit `--no-perf`) to populate this section.

11.8 Access Pattern Interactions with Data Structure Layout

The interaction between access pattern and memory layout explains much of the performance variation:

Sequential vs. random insert. Sequential insert favours structures with efficient append paths. All B-tree variants (matryoshka, TLX, Abseil) benefit because new keys land at the rightmost leaf. `std::set` benefits less because red-black rebalancing is oblivious to key order. The throughput ratio (seq/rand) at N=16,384 reveals how much each structure benefits from locality: matryoshka 1.42× vs. std::set on sequential, 1.09× on random.

Delete after bulk-load vs. interleaved. The `rand_delete` workload starts from a bulk-loaded (optimally packed) tree, giving every structure its best starting point. The `mixed` workload, by contrast, operates on a tree that is continuously modified, creating internal fragmentation. Structures that maintain good occupancy under churn (B-trees with merge/redistribute) degrade less between these workloads than structures with per-node allocation (`std::set`).

The 4 KiB page boundary. Matryoshka's 4 KiB page leaves are sized to match the OS page size, ensuring that navigating the CL sub-tree within a leaf never crosses a page boundary. TLX and Abseil leaves are smaller (<1 KiB), so multiple leaves may share a page—good for spatial locality of adjacent leaves, but each leaf may straddle two cache lines for the `memmove` operation. `std::set` nodes are scattered across the heap with no page-alignment guarantees.

SIMD and branch prediction. Matryoshka's SIMD search produces a bit mask rather than a conditional branch, making it prediction-friendly. The sorted-array B-trees (TLX, Abseil) use scalar binary search with $O(\log B)$ data-dependent branches per leaf, which the branch predictor struggles with for uniform random keys (50/50 taken probability at each comparison).

11.9 Proposed Additional Access Patterns

Several workloads not currently benchmarked would reveal different performance relationships:

Zipfian (skewed) insert
: A Zipfian distribution concentrates inserts on a small number of "hot" leaves. B-tree variants would benefit from cache-hot leaves; `std::set` would benefit from a cache-hot path of recently accessed nodes. Matryoshka's per-page CL sub-tree might show more frequent CL splits under concentrated load.

Range scan after insert
: Iterate over a range of k keys (e.g. $k = 1000$) after building the tree. Matryoshka's linked leaf pages and dense packing should excel; `std::set`'s in-order traversal via parent pointers would lag. This would highlight the spatial locality advantage of contiguous leaf storage.

Interleaved point lookup and insert
: A read-modify-write pattern ("contains then insert if absent") would test whether search and insert share cache-hot state. Matryoshka's search and insert paths share the same CL sub-tree navigation code, so a just-searched path remains cache-hot for the subsequent insert.

Large-key workload
: Keys longer than 4 bytes (e.g. 16- or 32-byte strings) would stress ART's strength (key-length-dependent, not N-dependent traversal) while increasing matryoshka's CL sub-node overhead (fewer keys per 64 B cache line). B-tree `memmove` cost would grow linearly with key size.

Delete-heavy with searches (YCSB-D)
: A workload where the tree shrinks from N to near-empty while servicing read queries. This would stress merge and redistribute paths, and test whether search throughput degrades as the tree becomes sparsely populated. Matryoshka's CL sub-node merge and page-level redistribute are designed for graceful degradation, but extreme sparsity (few keys spread across many pages) could hurt cache utilisation.

Bulk-load comparison
: Timing the bulk-load operation itself (currently untimed) would highlight structural differences: matryoshka distributes keys across CL sub-nodes within pages in a single bottom-up pass; TLX and Abseil use repeated insertion; `std::set` has no bulk-load optimisation.

11.10 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 sorted leaf array. At the largest dataset size (N=16,384), matryoshka's throughput is within 1.4× of the best B-tree competitor on insert-heavy workloads and 3.1× on delete-heavy workloads.

The primary cost of the nesting design is a constant-factor overhead per modification (CL sub-tree navigation), which the flat sorted-array B-trees avoid. This overhead is most visible at small N where the outer tree is shallow and the per-operation cost is dominated by in-leaf work. At large N, where the outer tree traversal and cache misses dominate, the nesting overhead is amortised and matryoshka's SIMD search and dense page layout become decisive advantages.

12 Improvements Since Initial Report

12.1 Superpage-Level Nesting (Implemented)

The nesting now extends to three levels: CL sub-nodes (64 B) within 4 KiB pages within 2 MiB superpages. Each superpage contains a B+ tree of page-level sub-nodes, with page-level internal nodes (681 separator keys, 682 children per 4 KiB page) routing searches to up to 510 page leaves. Maximum capacity per superpage: 510 × 855 ≈ 436K keys. This confines most operations to a single TLB entry and reduces outer-tree height. Enable via `mt_hierarchy_init_superpage`.

12.2 Wider SIMD: AVX2 and AVX-512 (Implemented)

Compile-time SIMD width selection via `-DMT_SIMD=avx2` or `-DMT_SIMD=avx512`. AVX2 (256-bit) processes 8 keys per comparison in CL leaf predecessor search, CL internal search, and outer internal node search. AVX-512 (512-bit) processes 16 keys per comparison using masked operations. Unaligned loads handle the 4-byte header offset within CL sub-nodes. SSE2 (128-bit, 4 keys) remains the baseline fallback.

12.3 Batch Insert and Delete API (Implemented)

`matryoshka_insert_batch(tree, keys, n)` and `matryoshka_delete_batch(tree, keys, n)` sort incoming keys, group them by target leaf, and amortise outer-tree traversal across each group. On page-full or underflow, the path is re-navigated for remaining keys. Both functions work with page leaves and superpages.

12.4 CL Sub-Tree Cache-Miss Optimisation: Fence Keys vs. Eytzinger (Implemented)

Instruction-level profiling (`perf record -e cache-misses` with `addr2line`) of the baseline matryoshka on `rand_insert` at N=16M identified three serial cache-miss hotspots within the CL sub-tree traversal:

% Miss	Source	Description
59.8%	`leaf.c:323`	`page->header.sub_height` — first access to the page header cache line
38.0%	`leaf.c:199`	`cl->nkeys` in `cl_inode_search` — loading the child CL internal node
76.3%*	`leaf.c:451`	`cl->nkeys < MT_CL_KEY_CAP` in `mt_page_insert` — loading the target CL leaf

*Percentage of `mt_page_insert`'s cache misses, not total.

These form a serial dependency chain: header → CL root internal → child CL node. Each load depends on data from the previous load, so no out-of-order execution or hardware prefetching can overlap them. The pointer-tagging optimisation (§??) already addresses hotspot 1 by prefetching the CL root from tagged pointers while still in the outer tree, but hotspots 2 and 3 remain. Two strategies were implemented and benchmarked head-to-head:

12.4.1 Strategy A: Fence Keys

Store the CL root internal's separator keys and child slot indices in the 32 spare bytes of the page header (previously `_reserved`). Since the page header is always loaded first (hotspot 1 is unavoidable), the fence data comes "for free"—the CL root internal can be skipped entirely for height-1 sub-trees with ≤ 6 separators.

Header layout.

```
/* Replaces _reserved[32] in mt_page_header_t */
int32_t fence_keys[6]; /* 24 bytes */
uint8_t fence_slots[7]; /* 7 bytes */
uint8_t nfence; /* 1 byte */
                          /* 32 bytes total */
```

When fence keys apply.

- Height 1, ≤ 7 children: fence keys fully resolve the CL leaf → **skip CL internal entirely**
- Height 2, root has ≤ 6 children: fence keys skip the root internal → saves one level
- Height 1, 8–13 children: `nfence=0`, transparent fallback to normal path

Maintenance. A `refresh_fence_keys()` helper copies $\min(\text{nkeys}, 6)$ keys and $\min(\text{nkeys} + 1, 7)$ child slots from the CL root internal into the header. It is called after bulk load, CL root split (in `mt_page_insert`), and CL root collapse (in `mt_page_delete`).

Page capacity. Unchanged (855 keys/page). Outer tree structure unchanged. Enable via `mt_hierarchy_init_fence`.

12.4.2 Strategy B: Eytzinger Dense BFS Layout

Fix the CL sub-tree to height ≤ 1 with a dense BFS layout. The root always occupies slot 1; children occupy slots $2, 3, \ldots, N+1$. Since child positions are arithmetic (not stored in the CL internal), all children can be prefetched simultaneously while the root cache line is still loading—breaking the dependency chain between hotspots 1 and 2.

CL internal type. A new 64-byte Eytzinger internal stores 15 separator keys with no `children[]` array (positions are implicit):

```
typedef struct mt_cl_inode_eytz {
    uint8_t type; /* MT_CL_INTERNAL */
    uint8_t nkeys; /* 0-15 */
    uint8_t nchildren; /* 1-16 */
    uint8_t _pad;
    int32_t keys[15]; /* 60 bytes */
} mt_cl_inode_eytz_t; /* 64 bytes total */
```

Key trade-off. With 16 children × 15 keys = 240 keys max per page, pages split at 240 keys instead of 855. This means ∼3.5× more pages in the outer tree, but each page operation is faster due to the eliminated serial miss. When a CL leaf splits within an Eytzinger page, the dense BFS invariant is restored by extracting all keys and rebuilding the layout—amortised to ∼16 key copies per insert.

Enable via `mt_hierarchy_init_eytzinger`.

12.4.3 Benchmark Results

Table 4: Fence keys vs. Eytzinger vs. baseline: `rand_insert` (ns/op, lower is better).

N	Baseline	Fence	Δ	Eytzinger	Δ
65,536	307	245	−20%	791	+158%
262,144	352	276	−21%	819	+133%
1,048,576	406	388	−4%	803	+98%
4,194,304	652	510	−22%	1,051	+61%
16,777,216	719	706	−2%	1,123	+56%

Table 5: Fence keys vs. Eytzinger vs. baseline: `search_after_churn` (ns/op).

N	Baseline	Fence	Δ	Eytzinger	Δ
65,536	232	193	−17%	228	−2%
16,777,216	785	668	−15%	771	−2%

12.4.4 Hardware Counter Analysis

`perf stat` on `rand_insert` at N=16M (P-core):

Table 6: Hardware counters: fence keys vs. Eytzinger vs. baseline.

Variant	Cache Miss (M)	L1d Miss (M)	Instr (B)	IPC
Baseline	393	145	12.6	0.65
Fence	384	130	12.7	0.69
Eytzinger	504	172	25.3	0.85

Fence keys reduced L1d misses by 10.3% and total cycles by 4.5%, with IPC improving from 0.65 to 0.69. The instruction count is essentially unchanged (+1%), confirming that the fence key fast path adds negligible overhead—it merely resolves the CL leaf from data already in the header cache line.

Eytzinger achieved dramatically better IPC (0.85 vs. 0.65) and a lower cache miss *rate* (43.8% vs. 66.6%), confirming that the mass-prefetch of all 16 children is working as intended: the CPU executes useful computation while prefetches resolve in parallel. However, the absolute miss count is 28% higher (504M vs. 393M) and the instruction count doubled (25.3B vs. 12.6B) due to the taller outer tree (3.5× more pages) and the $O(240)$ extract-rebuild on every CL leaf split.

12.4.5 Cache-Miss Profile Shift with Fence Keys

`perf record -e cache-misses` on `rand_insert` at N=16M:

Table 7: Cache-miss attribution: baseline vs. fence keys.

Function	Baseline	Fence	Change
`page_find_leaf`	51.6%	48.2%	−3.4 pp
of which `cl_inode_search`	31.7%	22.6%	−9.1 pp
`mt_page_insert`	25.3%	28.8%	+3.5 pp
`mt_inode_search`	14.2%	14.2%	∼0
`matryoshka_insert`	3.0%	3.1%	∼0

The fence keys reduced `cl_inode_search` cache misses by 9.1 percentage points—exactly the CL root internal loads being skipped for height-1 sub-trees with ≤ 6 separators. The residual 22.6% comes from: (1) height-2 pages where fence keys skip the root but a level-1 internal still requires loading; and (2) height-1 pages with 7–12 separators that exceed the 6-key fence capacity.

Instruction-level attribution of the remaining hotspots with fence keys:

`page_find_leaf` at `leaf.c:199` (35.6%)
: `int n = cl->nkeys` in `cl_inode_search`—loading the child CL internal after fence keys resolved the root. This fires on height-2 pages and height-1 pages with > 6 separators.

`mt_page_insert` at `leaf.c:62` (17.7% overall)
: `int lo = 0, hi = cl->nkeys` in `cl_leaf_lower_bound`—loading the target CL leaf to find the insertion point. This is the third miss in the serial chain: header → (fence skip root) → child CL leaf. It is now the **dominant remaining bottleneck**.

`mt_inode_search` at `inode.c:91` (13.6%)
: Binary search loop within outer tree internal nodes—standard outer-tree traversal cost, unaffected by CL-level optimisations.

12.4.6 Conclusions

1. **Fence keys are the clear winner.** 17–22% faster on insert and search at sizes fitting L3 cache (up to N=4M), with 2–15% improvement persisting at N=16M. Zero impact on page capacity or outer tree structure.
2. **Eytzinger is a negative trade-off.** Despite achieving better IPC (0.85 vs. 0.65) and a lower cache miss rate, the 3.5× page count increase (240 vs. 855 keys/page) and $O(240)$ extract-rebuild on every CL leaf split make it consistently slower—56–158% slower on insert across all sizes.
3. **The dominant remaining bottleneck** is loading the target CL leaf (`cl_leaf_lower_bound` at 17.7% of total cache misses). This is the irreducible third step in the serial chain that neither strategy can eliminate: the CL leaf's address is only known after searching the CL internal one level above.
4. **Fence keys hurt delete** by ∼15% due to the `refresh_fence_keys()` overhead after structural changes (merge, redistribute, root collapse). Future work could defer fence refresh to the next search, amortising the cost across multiple modifications.

12.5 Future: 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.